OpenLMIS Documentation

Release 3.0

VillageReach

Jun 24, 2017

Contents

OpenLMIS (Open Logistics Management Information System) is software for a shared, open source solution for managing medical commodity distribution in low- and middle-income countries. For more information, see OpenLMIS.org.

CHAPTER 1

Contents:

Architecture

As of OpenLMIS v3, the architecture has transitioned to (micro) services fulfilling RESTful (HTTP) API requests from a modularized Reference UI. Extension mechanisms in addition to microservices and UI modules further allow for components of the architecture to be customized without the need for the community to fork the code base:

- UI modules give flexibility in creating new user experiences or changing existing ones
- Extension Points & Modules - allows Service functionality to be modified
- Extra Data - allows for extensions to store data with existing components

Combined these components allow the OpenLMIS community to customize and contribute to a shared LMIS.

Docker

Docker Engine and Docker Compose is utilized throughout the tech stack to provide consistent builds, quicken environment setup and ensure that there are clean boundaries between components. Each deployable component is versioned and published as a Docker Image to the public Docker Hub. From this repository of ready-to-run images on Docker Hub anyone may pull the image down to run the component.

Development environments are typically started by running a single Service or UI module's development docker compose. Using docker compose allows the component's author to specify the tooling and test runtime (e.g. PostgreSQL) that's needed to compile, test and build and package the production docker image that all implementation's are intended to use.

After a production docker image is produced, docker compose is used once again in the Reference Distribution to combine the desired deployment images with the needed configuration to produce an OpenLMIS deployment.

UI

OpenLMIS-UI Application Structure

The OpenLMIS-UI is an Angular V1 application that communicates with OpenLMIS Services to create a human-centered user-interface to the OpenLMIS application. This UI application is meant to be extendable, and provide a framework for developers to create a consistent experience.

This UI aims to support accessiblity, consistency, and i18n compliance.

This is a high-level overview of how the OpenLMIS-UI functions, for more details please see:

- UI build process documentation
- UI extention guide
- UI coding conventions

Page Load

The entire OpenLMIS-UI is a single page application that is loaded in a single HTML page. The page loads a single CSS and a single Javascript file that are generated from the build process. A loading icon is shown while the HTML page waits for javascript file to load and run.

When the HTML page is loaded, an Angular applicaiton called 'openlmis' is started, which requires all other Javascript modules that make up the OpenLMIS-UI application. **No module should ever require 'openlmis' or add a module directly to it.** The 'openlmis' AngularJS module is dynamically built during the build process.

Page Views

The OpenLMIS-UI is state driven, meaning the browser's URL determines what is displayed on the screen (with some exceptions). Once the application starts, the browser's current URL is parsed by UI-Router; this populates the application view and retrieves data from the OpenLMIS Services.

Not all page views are directly defined as a page route. Some modules will provide functionality that intercepts page views or data requests to the OpenLMIS services, and provide custom logic. Examples of this are authentication and offline modules.

Authentication

Authentication with the OpenLMIS services is handled by the openlmis-auth-ui by intercepting UI-Router and $HTTP requests. The access tokens used by OpenLMIS are managed by interceptors, and are automatically appended to all requests to the OpenLMIS services. **No OpenLMIS-UI modules should need to include openlmis-auth because it's included by the OpenLMIS-UI application.**

Offline and Low-Bandwidth

The OpenLMIS-UI supports users who are offline or have low-bandwidth connections by focusing on the browser's AppCache and localstorage technologies.

Build Process

The OpenLMIS-UI functions as a single-page application that is created through a Grunt-based build process.

This document details the different workflows a developer might use while working on the OpenLMIS-UI.

Standard Build

The most standard way to build the OpenLMIS-UI is to run `grunt build` which will concatinate, and compile the OpenLMIS-UI into a working set of front-end assets that are ready for development. When run, the following high level tasks will be exectuted:

- clean the `build` directory
- -style check files in `src`-
- create OpenLMIS application in `build/webapp`
- run unit test in `src` against `build/webapp/openlmis.js`
- create OpenLMIS styleguide in `build/styleguide`
- create OpenLMIS Javascript Documentation in `build/docs`

Build Flags

There are a number of settings that can be set when building the OpenLMIS-UI.

Production

Running a command with the `--production` flag will make the grunt build command compress all file types, getting the UI ready for production rather than development.

```
grunt build --production
```

Unit tests

Running unit tests can be skipped by adding the `--noTest` flag to the command.

```
grunt build --noTest
```

Styleguide

Building the styleguide can be skipped by adding `--noStyleguide` to the command.

```
grunt build --noStyleguide
```

Javascript Documentation

Choose not to build the Javascipt Documentation by add `--noDocs` to the command.

```
grunt build --noDocs
```

Application

Build only the OpenLMIS application, and skip all tests by adding `--appOnly` to the command.

```
grunt build --appOnly
```

Service Paths

Since the OpenLMIS Services might be in a different location than the OpenLMIS-UI, you can change the path to the OpenLMIS server location through a build flag. There are actually hooks to use OpenLMIS services hosted in different locations.

OpenLMIS Server path

`--openlmisServerURL` is the path to the main OpenLMIS server. This URL will be prepended to all other service paths, unless the service path starts with `http`

```
grunt build --openlmisServerURL=http://somewhere.com/
```

Auth Service path

```
grunt build --authServiceURL=/where/to/find/auth
```

Requisition Service path

```
grunt build --requisitionServiceURL=http://requisitions.are/here
```

Proxy Service

If the OpenLMIS-UI is located at a different root domain than the OpenLMIS Services, the browser will not run the OpenLMIS-UI because of CORS errors. Since it is not always practical to set up CORs on a development instance of OpenLMIS — while developing the UI a developer can use the `--addProxyService` a command, which will prepend a proxy service location to any OpenLMIS Server URL.

A developer will also need to start a proxy service on the development server by running `grunt serve --addProxyService`

```
// sets the openlmisServerURL to 'http://127.0.0.1:3000/http://over/there'
grunt build --openlmisServerURL=http://over/there --addProxyService

// starts a proxy server at http://127.0.0.1:3000
grunt serve --addProxyService
```

Automatic Building

When working on the OpenLMIS-UI it's convient to have the entire UI rebuilt when changes are made to the source files. This is achieved by running `grunt watch`. Everytime a file change is detected in `src/` grunt will re-run the build process with all the flags from when `grunt watch` was run.

It is recommended that one run grunt watch to speed up application development

```
grunt watch --openlmisServerURL=http://where.openlmis.is/
```

Test Driven Development

A test driven development server can be run, where each time the `build/openlmis.js` or any unit test file is change will trigger the unit tests to be automatically rerun. This is a great way to karma and phantomjs tests because the test browser is never closed, meaning tests will run much faster.

```
grunt karma:tdd
```

Debugging Test Driven Development

Debugging karma tests can be difficult, but to ease the process you can debug tests in your machines local browser by visiting Karma's local server. When debugging you can set break points in unit tests or OpenLMIS-UI source files. Instructions for capturing a browser manually are on this page.

When karma starts, the console will output the karma server location, which will look like:

```
01 01 1970 0:0:0.0:INFO [karma]: Karma v0.13.22 server started at http://
↪localhost:9876/
```

Then take your browser and go to `http://localhost:9876` (which might change depending on if a process is already using the port).

Once the page loads you will see the karma tests are run again, but this time there are two sets of tests running (once in PhantomJS, and once in your browser). In the right hand corner there is a debug button, which will open a new window where you can view all the test output in the console and set breakpoints using your browser's developer tools.

UI Extention Architecture and Guide

This document outlines how the OpenLMIS UI supports extension and customization. The OpenLMIS UI is an AngularJS browser application that is built from multiple sources into a single app. The OpenLMIS UI works with the OpenLMIS APIs using RESTful HTTP requests. Below are explanations of the core extension components, along with brief examples of how to extend sections of the UI. To learn more about the files that are produced by the OpenLMIS-UI build process, see the build process documentation.

Core technologies:

- Javascript and AngularJS v1.6
- Docker and NodeJS for compiling a single page application from multiple sources
- Sass variables and mixins for easy style extention
- UI-Router for page definition
- Extention service for adding new functionality to existing components (in development)

Docker Architecture

To create a modular single page application, we are using Docker and NodeJS to build multiple git repositories together into the working UI layer for OpenLMIS. The pattern that the OpenLMIS-UI follows is extremely similar to the micro-services architecture used in OpenLMIS, with the large difference being that the OpenLMIS-UI is compiled and does not 'discover' UI components.

There are 3 types of Docker images that need to be combined to make a functional UI

- A **tooling image** that stores core components which can compile the UI (see OpenLMIS/Dev-UI)
- **Source code images** that contain Javascript, HTML, CSS/SCSS, and other assets that make up functional sections of the OpenLMIS-UI (See OpenLMIS-Requisition-UI)
- A **publishing image,** that compiles the source code images and builds a working web server that is included into the OpenLMIS Server. Small adjustments like branding should be made in the publishing image. (See OpenLMIS/Reference-UI)

When the OpenLMIS-UI is compiled, the source code images will overwrite each other, according to the order specified in the config.json file (see Dev-UI Documentation). This makes it possible to override assets or sections of behavior in a simple process.

Note: The decision to use implicit over-writing of files rather than an explicit configuration came from Javascript's global name space which would cause problems if two Javascript objects had the same name.

Example: There are 2 images being included into a publishing image, and they have files at the following paths:

OpenLMIS-UI-Common

- *src/common/* logo.png
- *src/common/* header.scss

OpenLMIS-Requisition-UI

- *src/requisition/* requisition.js
- *src/requisition/* requisition.routes.js

Example-UI-Distribution

- *src/common/* logo.png
- *src/requisition/* requisition.routes.js

When comiled the following sources will be used:

- **Example-UI-Distribution/** *src/common/* logo.png (new logo)
- **OpenLMIS-UI-Common/** *src/common/* header.scss
- **OpenLMIS-Requisition-UI/** *src/requisition/* requisition.js
- **Example-UI-Distribution/** *src/requisition/* requisition.routes.js (changed requisition pages that are available)

Guidelines and Examples

The following are brief examples of sections of the OpenLMIS-UI that can be extended, which are organized by the goal an implementation is trying to achieve. The extention goals the OpenLMIS-UI supports are:

- **Modifying content** such as specific message strings, images, or specific parts of HTML markup
- **Global styles** such as CSS presentation or singular directives

- **Pages** meaning adding new pages to the OpenLMIS-UI, completely replacing an existing screen, or adding additional content to a specific screen.

NOTE: Adding logic to existing Javascript classes is not currently documented, but is a need we are aware of.

Modifying content

There are multiple places that the OpenLMIS-UI allows an implementer to customize content.

Messages

Messages are translateable pieces of content that are a part of the OpenLMIS-UI. The best way to update a message is by configuring the OpenLMIS-UI to use your implementation's Transifex settings and editing the message directly in Transifex. What Transifex does is replace a message key, which is defined in the UI, with a human readable message. *This is how the OpenLMIS-UI supports multiple languages.*

See the example below:

```
<!-- HTML similar to this, will be updated by Transifex once the OpenLMIS-UI is built␣
↪and run in a browser -->
<p>{{ 'example.instructions' | message }}</p>

<!-- The above message will be displayed to a person using the OpenLMIS-UI as: -->
<p>These are some instructions</p>
```

The strategy in the OpenLMIS-UI is to not reuse specific message keys, and let the tools in Transifex group messages that are the same together into a single spot for translation. This allows an implementer to further customize small messages such as 'Search' to 'Search Facilities' if the needs of their implementation require more specificity.

If there are large message string changes, it is possible for an implementer to replace the `messages_en.json` file, and make the changes that are needed. **This is possible as a last resort but not recommended.**

Images

Individual images are also easy to change, either by overriding the CSS style (see below) or 'implicitly' replacing the image. As mentioned earlier, all files overwrite each other implicitly, so to replace the default logo (for example) just replace the logo.png file in the publishing image.

HTML Layouts

There might be places where an implementor needs to completely change the layout of a page or element. An example of this might be adding an additional header, or a paragraph of instructions. This is simple because of AngularJS's markup pattern and the implicit file replacement.

Let's consider wanting to add a large footer element to the OpenLMIS-UI (which might contain help information or an emergency phone number). To do this, an implementer would replace the main index.html file and add a CSS file with the styling needed.

The HTML markup in this new index.html file might look like:

```
<!-- NOTE: Dropped the HTML and HEAD sections -->
<body >
    <header ng-if="userIsAuthenticated">
```

```
      <div ng-include="'openlmis-header/header.html'" offline ng-class="{'isOffline
↪':isOffline}" ></div>
      <div class="navbar">
        <openlmis-navigation class="nav navbar-nav"></openlmis-navigation>
        <openlmis-locale class="navbar-right locale-container"></openlmis-locale>
      </div>
    </header>
    <div ui-view></div>
    <!-- This is new -->
    <footer>
    ... phone numbers and additional information here ...
    </footer>
    <!-- /new -->
  </body>
```

NOTE: The main `index.html` was designed to be extremely minimal so that implementations could make huge changes quickly. Other HTML files within OpenLMIS follow this same paradigm of easy to replace HTML.

Global Styles

Global styles refer to CSS styles and Javascript interactions that are consistently applied throughout the OpenLMIS-UI. These designs and styles are purposefully implemented on extremely vanilla HTML to promote consistency and to make it easy for an implementer to quickly change interaction patterns across the entire UI. See the coding conventions for more information about how this is achieved.

CSS and SASS

There should be no special method of removing or explicitly overriding CSS — an implementer should simply use more specific CSS rules to override a CSS style. The CSS coding conventions stress using shallow selectors and avoiding the `!important` selector, so that its easy for an implementor to make style changes in a publishing image.

```
// In general, overriding a style is as simple as adding a higher level selector like
↪'body'
// Here is how an implementer might make all the links in the OpenLMIS-UI underlined

body a {
  text-decoration: underline;
}
```

In addition to keeping the CSS simple, the OpenLMIS-UI uses SASS variables to implement colors and spacing across the UI, which means changing a color is as simple as declaring a variable. The OpenLMIS-UI's sass varibles follow a semantic naming pattern, and `!default` variables are implemented in smaller files, which make it easy to update patterns. A full list of variables in an implementation is availabe in the generated OpenLMIS-UI styleguide.

To update Sass variables in a publishing image, create a file `src/variables.scss` and declare variables without `!default`

```
$brand-primary: #FDFB50; // Turned most branding elements a disgusting yellow
$brand-danger: #AE4442; // Made error elements use a darker more brown red
```

AngularJS Directives and Components

Within the OpenLMIS-UI there are many optimizations that use AngularJS directives to keep HTML markup DRY and provide a consistent experience. The core OpenLMIS-UI attempts to provide a simple, accessible, and usable experience – but for some implementations these patterns might need to be changed or updated. See the OpenLMIS-UI coding conventions for a discussion of the difference between a directive and a component.

The best way to change these patterns is to implicitly replace the file where the directive is created.

NOTE: Changing directives is tricky, and a simple example will be added to this document ... eventually ...

Pages

Pages are the primary unit of any application, and the OpenLMIS-UI has support for adding new pages, replacing existing pages, and adding content to existing pages.

New Pages

To add a new page to OpenLMIS-UI an implementer needs to register a page with UI-Router. The OpenLMIS-Navigation directive can expose pages registered with UI-Router, which is used within the OpenLMIS-UI in the main header navigation, meaning that new pages are exposed to a user in the OpenLMIS-UI. See the UI coding conventions for specifics on how to add a route, and the configuration options available.

Here is a simple example.

```
// Consider adding a view to see requisitions on a map
angular.module('custom-module').config(function($stateProvider){
  $stateProvider.state('requisitions.map', {
    url: '/map',
    showInNavigation: true,
    controller: 'MyCustomMapController',
    templateUrl: 'map/page.html'
  });
});
```

NOTE: Adding new pages to the OpenLMIS-UI should mostly be done in source code images, if the page adds functionality. If the page is implementation specific it sould be added in a publishing image.

Replacing an Existing Screen

We expect that some implementations will need to make small changes to existing pages within the OpenLMIS-UI that go beyond the previous extention techniques mentiond. The most simple method of replacing a screen is implicitly replacing the `*.routes.js` file where the route is defined, as these files are mostly configuration and shouldn't contain complex logic.

A more complex alternative is to modify the UI-Router configuration at run-time, which would allow for very nuanced changes that require their own unique unit tests. The following example shows how to change a page's template depending on a user's access rights. **NOTE:** This is an invasive techinque, and should only be done if all other extention methods don't work

```
angular.module('custom-module').run(function($state){
  var state = $state.get('home'); // get the current home page state
  state.view.templateUrl = 'new_tempalte.html';
```

```
  // gotta figure out how to change this...
});
```

Extending a Screen

This functionality is still underdevelopment, see OLMIS-1682: Extention directive to allow insertion of UI components for implementation details.

Components

How to use Component Documentation

Logging in with the Live Documentation

The live documentation links connect directly to our Swagger docs on our CI server. To use the API you'll first need to get an access token from the Auth service, and then you'll need to give that token when using one of the RESTful operations.

Obtaining an access token:

1. goto the Auth service's POST /api/oauth/token
2. click on `Authorize` in the top right of the page
3. in the box that has popped-up, enter username `user-client` and password `changeme`
4. click `Authorize` under `password`
5. enter the username `administrator` and password `password`
6. click `Try it out!`
7. In the `Response body` box, copy the UUID. e.g. `"access_token": "a93bcab7-aaf5-43fe-9301-76c526698898"` copy `a93bcab7-aaf5-43fe-9301-76c526698898` to use later.
8. Paste the UUID you just copied into any endpoint's `access_token` field.

Auth

Security credentials, Authentication and Authorization. Uses OAuth2.

- Auth Service
- Auth UI
- Auth ERD
- Live Documentation for Auth API
- Static Documentation for Auth API

Fulfillment Service

Includes the basics of fulfillment.

- Fulfillment Service
- Fulfillment UI
- Fulfillment ERD
- Live Documentation for Fulfillment API
- Static Documentation for Fulfillment API

Notification Service

Notifying users when their attention is needed.

- Live Documentation for Notification API
- Static Documentation for Notification API

Reference Data Service

Provides the reference data for the rest of the processes: facilities, programs, products, etc.

- Reference Data Service
- Reference Data UI
- Reference Data ERD
- Live Documentation for Reference Data API
- Static Documentation for Reference Data API

Reference UI

The Reference UI compiles together all the assets that make up the OpenLMIS-UI. See the build process documentation to understand exactly how the UI is compiled.

- UI Styleguide
- Javascript Documentation
- Reference UI
- UI Layout
- UI Components
- Dev UI

Requisition Service

Requisition (pull) based replenishment process.

- Requisition Service
- Requisition UI

- Requisition ERD
- Live Documentation for Requisition API
- Static Documentation for Requisition API

Stock Management Service

Electronic stock cards.

- Stock Management Service
- Stock Management UI
- Stock Management ERD
- Live Documentation for Stock Management API
- Static Documentation for Stock Management API

Conventions

The License Header

Each java or javascript file in the codebase should be annotated with the proper copyright header. This header should be also applied to singnificant html files.

We use checkstyle to check for it being present in Java files. We also check for it during our Grunt build in javascript files.

The current copyright header format can be found [here.] (https://raw.githubusercontent.com/OpenLMIS/openlmis-ref-distro/master/LICENSE-HEADER)

Replace the year and holder with appropriate holder, for example:

```
Copyright © 2017 VillageReach
```

Service Conventions

OpenLMIS Service Style Guide

This is a WIP as a style guide for an Independent Service. Clones of this file should reference this definition.

For the original style guide please see: https://github.com/OpenLMIS/open-lmis/blob/master/STYLE-GUIDE.md

---

Java

OpenLMIS has adopted the Google Java Styleguide. These checks are *mostly* encoded in Checkstyle and should be enforced for all contributions.

Some additional guidance:

- Try to keep the number of packages to a minimum. An Independent Service's Java code should generally all be in one package under `org.openlmis` (e.g. `org.openlmis.requisition`).

- Sub-packages below that should generally follow layered-architecture conventions; most (if not all) classes should fit in these four: `domain`, `repository`, `service`, `web`. To give specific guidance:
  - Things that do not strictly deal with the domain should NOT go in the `domain` package.
  - Serializers/Deserializers of domain classes should go under `domain`, since they have knowledge of domain object details.
  - DTO classes, belonging to serialization/deserialization for endpoints, should go under `web`.
  - Exception classes should go with the classes that throw the exception.
  - We do not want separate sub-packages called `exception`, `dto`, `serializer` for these purposes.
- When wanting to convert a domain object to/from a DTO, define Exporter/Importer interfaces for the domain object, and export/import methods in the domain that use the interface methods. Then create a DTO class that implements the interface methods. (See Right and RightDto for details.)
  - Additionally, when Exporter/Importer interfaces reference relationships to other domain objects, their Exporter/Importer interfaces should also be used, not DTOs. (See example.)
- Even though the no-argument constructor is required by Hibernate for entity objects, do not use it for object construction; use provided constructors or static factory methods. If one does not exist, create one using common sense parameters.

RESTful Interface Design & Documentation

Designing and documenting

Note: many of these guidelines come from Best Practices for Designing a Pragmatic RESTful API.

- Result filtering, sorting and searching should be done by query parameters. Details
- Return a resource representation after a create/update. Details
- Use camelCase (vs. snake_case) for names, since we are using Java and JSON. Details
- Don't use response envelopes as default (if not using Spring Data REST). Details
- Use JSON encoded bodies for create/update. Details
- Use a clear and consistent error payload. Details
- Use the HTTP status codes effectively. Details
- Resource names should be pluralized and consistent. e.g. prefer `requisitions`, never `requisition`.
- A PUT on a single resource (e.g. PUT /facilities/{id}) is not strictly an update; if the resource does not exist, one should be created using the specified identity (assuming the identity is a valid UUID).
- Exceptions, being thrown in exceptional circumstances (according to *Effective Java* by Joshua Bloch), should return 500-level HTTP codes from REST calls.
- Not all domain objects in the services need to be exposed as REST resources. Care should be taken to design the endpoints in a way that makes sense for clients. Examples:
  - `RoleAssignments` are managed under the users resource. Clients just care that users have roles; they do not care about the mapping.
  - `RequisitionGroupProgramSchedules` are managed under the requisitionGroups resource. Clients just care that requisition groups have schedules (based on program).
- RESTful endpoints that simply wish to return a JSON value (boolean, number, string) should wrap that value in a JSON object, with the value assigned to the property "result". (e.g. `{ "result": true }`)

  - Note: this is to ensure compliance with all JSON parsers, especially ones that adhere to RFC4627, which do not consider JSON values to be valid JSON. See the discussion here.

- When giving names to resources in the APIs, if it is a UUID, its name should have a suffix of "Id" to show that. (e.g. `/api/users/{userId}/fulfillmentFacilities` has query parameter `rightId` to get by right UUID.)

We use RAML (0.8) to document our RESTful APIs, which are then converted into HTML for static API documentation or Swagger UI for live documentation. Some guidelines for defining APIs in RAML:

- JSON schemas for the RAML should be defined in a separate JSON file, and placed in a `schemas` subfolder in relation to the RAML file. These JSON schema files would then be referenced in the RAML file like this (using role as an example):

```
- role: !include schemas/role.json

- roleArray: |
  {
    "type": "array",
    "items": { "type": "object", "$ref": "schemas/role.json" }
  }
```

  - (Note: this practice has been established because RAML 0.8 cannot define an array of a JSON schema for a request/response body (details). If the project moves to the RAML 1.0 spec and our RAML testing tool adds support for RAML 1.0, this practice might be revised.)

Pagination

Many of the GET endpoints that return *collections* should be paginated at the API level. We use the following guidelines for RESTful JSON pagination:

- Pagination options are done by *query* paramaters. i.e. use `/api/someResources?page=2` and not `/api/someResources/page/2`.
- When an endpoint is paginated, and the pagination options are *not* given, then we return the full collection. i.e. a single page with every possible instance of that resource. It's therefore up to the client to use collection endpoints responsibly and not over-load the backend.
- A paginated resource that has no items returns a single page, with it's `content` attribute as empty.
- Resource's which only ever return a single identified item are *not* paginated.
- For Java Service's the query parameters should be defined by a Pageable and the response should be a Page.

Example Request (note that page is zero-based):

```
GET /api/requisitions/search?page=0&size=5&access_token=<sometoken>
```

Example Response:

```
{
  "content": [
    {
    ...
    }
  ],
  "totalElements": 13,
  "totalPages": 3,
  "last": false,
```

```
  "numberOfElements": 5,
  "first": true,
  "sort": null,
  "size": 5,
  "number": 0
}
```

Postgres Database

For guidelines on how to write schema migrations using Flyway, see Writing Schema Migrations (Using Flyway).

- Each Independent Service should store its tables in its own schema. The convention is to use the Service's name as the schema. e.g. The Requisition Service uses the `requisition` schema
- Tables, Columns, constraints etc should be all lower case.
- Table names should be pluralized. This is to avoid *most* used words. e.g. orders instead of order
- Table names with multiple words should be snake_case.
- Column names with multiple words should be merged together. e.g. `getFirstName()` would map to `firstname`
- Columns of type uuid should end in 'id', including foreign keys.

RBAC (Roles & Rights) Naming Conventions

- Names for rights in the system should follow a RESOURCE_ACTION pattern and should be all uppercase, e.g. REQUISITION_CREATE, or FACILITIES_MANAGE. This is so all of the rights of a certain resource can be ordered together (REQUISITION_CREATE, REQUISITION_AUTHORIZE, etc.).

i18n (Localization)

Transifex and the Build Process

OpenLMIS v3 uses Transifex for translating message strings so that the product can be used in multiple languages. The build process of each OpenLMIS service contains a step to sync message property files with a corresponding Transifex project. Care should be taken when managing keys in these files and pushing them to Transifex.

- If message keys are added to the property file, they will be added to the Transifex project, where they are now available to be translated.
- If message keys or strings are modified in the property file, any translations for them will be lost and have to be re-translated.
- If message keys are removed in the property file, they will be removed from the Transifex project. If they are re-added later, any translations for them will be lost and have to be re-translated.

Naming Conventions

These naming conventions will be applicable for the messages property files.

- Keys for the messages property files should follow a hierarchy. However, since there is no official hierarchy support for property files, keys should follow a naming convention of most to least significant.

- Key hierarchy should be delimited with a period (.).
- The first portion of the key should be the name of the Independent Service.
- The second portion of the key should indicate the type of message; error for error messages, message for anything not an error.
- The third and following portions will further describe the key.
- Portions of keys that don't have hierarchy, e.g. `a.b.code.invalidLength` and `a.b.code.invalidFormat`, should use camelCase.
- Keys should not include hyphens or other punctuation.

Examples:

- `requisition.error.product.code.invalid` - an alternative could be `requisition.error.productCode.invalid` if code is not a sub-section of product.
- `requisition.message.requisition.created` - requisition successfully created.
- `referenceData.error.facility.notFound` - facility not found.

Note: UI-related keys (labels, buttons, etc.) are not addressed here, as they would be owned by the UI, and not the Independent Service.

Testing

See the Testing Guide.

Docker

Everything deployed in the reference distribution needs to be a Docker container. Official OpenLMIS containers are made from their respective containers that are published for all to see on our Docker Hub.

- Dockerfile (Image) best practices
- Keep Images portable & one-command focused. You should be comfortable publishing these images publicly and openly to the DockerHub.
- Keep Containers ephemeral. You shouldn't have to worry about throwing one away and starting a new one.
- Utilize docker compose to launch containers as services and map resources
- An OpenLMIS Service should be published in one image found on Docker Hub
- Services and Infrastructure that the OpenLMIS tech committee owns are published under the "openlmis" namespace of docker and on the Docker Hub.
- Avoid Docker Host Mounting, as this doesn't work well when deploying to remote hosts (e.g. in CI/CD)

Gradle Build

Pertaining to the build process performed by Gradle.

- Anything generated by the Gradle build process should go under the `build` folder (nothing generated should be in the `src` folder).

Logging

Each Service includes the SLF4J library for generating logging messages. Each Service should be forwarding these log statements to a remote logging container. The Service's logging configuration should indicate the name of the service the logging statement comes from and should be in UTC.

What generally should be logged:

- DEBUG - should be used to provide more information to developers attempting to debug what happened. e.g. bad user input, constraint violations, etc
- INFO - to log processing progress. If the progress is for a developer to understand what went wrong, use DEBUG. This tends to be more useful for performance monitoring and remote production debugging after a client's installation has failed.

Less used:

- FATAL - is reserved for programming errors or system conditions that resulted in the application (Service) terminating. Developers should not be using this directly, and instead use ERROR.
- ERROR - is reserved for programming conditions or system conditions that would have resulted in the Service terminating, however some safety oriented code caught the condition and made it safe. This should be reserved for a global Service level handler that will convert all Exceptions into a HTTP 5xx level exception.

Audit Logging

Services use JaVers to log changes made throughout the system. The audits logs for individual resources should be exposed via endpoints which look as follows:

```
/api/someResources/{id}/auditLog
```

Just as with other paginated endpoints, these requests may be filtered via *page* and *size* query paramaters: `/api/someResources?page=0&size=10`

The returned log may additionally be filtered by *author* and *changedPropertyName* query paramaters. The later specifies that only changes made by a given user should be returned, whereas the later dictates that only changes related to the named property should be shown.

Each `/api/someResources/{id}/auditLog` endpoint should return a 404 error if and only if the specified {id} does not exist. In cases where the resource id exists but lacks an associated audit log, an empty array representing the empty audit should be returned.

Within production services, the response bodies returned by these endpoints should correspond to the JSON schema defined by *auditLogEntryArray* within */resources/api-definition.yaml*. It is recognized and accepted that this differs from the schema intended for use by other collections throughout the system. Specifically, whereas other collections which support paginated requests are expected to return pagination-related metadata (eg: "totalElements," "totalPages") within their response bodies, the responses proffered by /auditLog endpoints do not retur pagination related data.

Testing Guide

This guide is intended to layout the general automated test strategy for OpenLMIS.

Test Strategy

OpenLMIS, like many software projects, relies on testing to guide development and prevent regressions. To effect this we've adopted a standard set of tools to write and execute our tests, and categorize them to understand what types of tests we have, who writes them, when they're written, run, and where they live.

Types of Tests

The following test categories have been identified for use in OpenLMIS. As illustrated in this great slide deck, we expect the effort/number of tests in each category to reflect the test pyramid:

1. *Unit*
2. *Integration*
3. *Component*
4. *Contract*
5. *End-to-End*

Unit Tests

- Who: written by code-author during implementation
- What: the smallest unit (e.g. one piece of a model's behavior, a function, etc)
- When: at build time, should be /fast/ and targeted - I can run just a portion of the test suite
- Where: Reside inside a service, next to unit under test. Generally able to access package-private scope
- Why: to test fundamental pieces/functionality, helps guide and document design and refactors, protects against regression

Unit Test Examples

- **Every single test should be independent and isolated. Unit test shouldn't depend on another unit test.**

  DO NOT:

```
List<Item> list = new ArrayList<>();

@Test
public void shouldContainOneElementWhenFirstElementisAdded() {
  Item item = new Item();
  list.add(item);
  assertEquals(1, list.size());
}

@Test
public void shouldContainTwoElementsWhenNextElementIsAdded() {
  Item item = new Item();
  list.add(item);
  assertEquals(2, list.size());
}
```

- **One behavior should be tested in just one unit test.**

  DO NOT:

```
@Test
public void shouldNotBeAdultAndShouldNotBeAbleToRunForPresidentWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);

  boolean isAbleToRunForPresident = electionsService.isAbleToRunForPresident(age)
  assertFalse(isAbleToRunForPresident);
}
```

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}

@Test
public void shouldNotBeAbleToRunForPresidentWhenAgeBelow18() {
  int age = 17;
  boolean isAbleToRunForPresident = electionsService.isAbleToRunForPresident(age)
  assertFalse(isAbleToRunForPresident);
}
```

- **Every unit test should have at least one assertion.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
}
```

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **Don't make unnecessary assertions. Don't assert mocked behavior, avoid assertions that check the exact same thing as another unit test.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  assertEquals(17, age);
```

```
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **Unit test has to be independent from external resources (i.e. don't connect with databases or servers)**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  String uri = String.format("http://127.0.0.1:8080/age/", HOST, PORT);
  HttpPost httpPost = new HttpPost(uri);
  HttpResponse response = getHttpClient().execute(httpPost);
  assertEquals(HttpStatus.ORDINAL_200_OK, response.getStatusLine().getStatusCode());
}
```

- **Unit test shouldn't test Spring Contexts. Integration tests are better for this purpose.**

  DO NOT:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/services-test-config.xml"})
public class MyServiceTest implements ApplicationContextAware
{

  @Autowired
  MyService service;
  ...
    @Override
    public void setApplicationContext(ApplicationContext context) throws␣
↪BeansException
    {
        // something with the context here
    }
}
```

- **Test method name should clearly indicate what is being tested and what is the expected output and condition. The "should - when" pattern should be used in the name.**

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  ...
}
```

  DO NOT:

```
@Test
public void firstTest() {
  ...
}

@Test
public void testIsNotAdult() {
  ...
}
```

- **Unit test should be repeatable - each run should yield the same result.**

DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = randomGenerator.nextInt(100);
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **You should remember about intializing and cleaning each global state between test runs.**

  DO:

```
@Mock
private AgeService ageService;
private age;

@Before
public void init() {
  age = 18;
  when(ageService.isAdult(age)).thenReturn(true);
}

@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  boolean isAdult = ageService.isAdult(age);
  assertTrue(isAdult);
}
```

- **Test should run fast. When we have hundreds of tests we just don't want to wait several minutes till all tests pass.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  sleep(1000);
  boolean isAdult = ageService.isAdult(age);
  sleep(1000);
  assertFalse(isAdult);
}
```

Integration Tests

- Who: Code author during implementation
- What: Test basic operation of a service to persistent storage or a service to another service. When another service is required, a test-double should be used, not the actual service.
- When: As explicitly asked for, these tests are typically slower and therefore need to be kept separate from build to not slow development. Will be run in CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Ensures that the basic pathways to a service's external run-time dependancies work. e.g. that a db schema supports the ORM, or a non-responsive service call is gracefully handled.

For testing controllers, they are divided up into unit and integration tests. The controller unit tests will be testing the logic in the controller, while the integration tests will be mostly testing serialization/deserialization (and therefore do not need to test all code paths). In both cases, the underlying services and repositories are mocked.

Component Tests

- Who: Code author during implementation
- What: Test more complex operations in a service. When another service is required, a test-double should be used, not the actual service.
- When: As explicitly asked for, these tests are typically slower and therefore need to be kept separate from build to not slow development. Will be run in CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Tests interactions between components in a service are working as expected.

These are not integration tests, which strictly test the integration between the service and an external dependency. These test the interactions between components in a service are working correctly. While integration tests just test the basic pathways are working, component tests verify that, based on input, the output matches what is expected.

These are not contract tests, which are more oriented towards business requirements, but are more technical in nature. The contract tests will make certain assumptions about components, and these tests make sure those assumptions are tested.

Contract Tests

- Who: Code author during implementation, with input from BA/QA.
- What: Enforces contracts between and to services.
- When: Ran in CI.
- Where: Reside inside separate repository: openlmis-contract-tests.
- Why: Tests multiple services working together, testing contracts that a Service both provides as well as the requirements a dependant has.

The main difference between contract and integration tests: In contract tests, all the services under test are *real*, meaning that they will be processing requests and sending responses. Test doubles, mocking, stubbing should not be a part of contract tests.

Refer to this doc for examples of how to write contract tests.

End-to-End Tests

- Who: QA / developer with input from BA.
- What: Typical/core business scenarios.
- When: Ran in CI.
- Where: Resides in seperate repository.
- Why: Ensures all the pieces are working together to carry-out a business scenario. Helps ensure end-users can achieve their goals.

Testing services dependent on external APIs

OpenLMIS is using WireMock for mocking web services. An example integration test can be found here: https://github.com/OpenLMIS/openlmis-example/blob/master/src/test/java/org/openlmis/example/WeatherServiceTest.java

The stub mappings which are served by WireMock's HTTP server are placed under *src/test/resources/mappings* and _src/test/resources/__files_ For instructions on how to create them please refer to http://wiremock.org/record-playback.html

Testing Tools

- spring-boot-starter-test
  - Spring Boot Test
  - JUnit
  - Mockito
  - Hamcrest
- WireMock
- REST Assured
- raml-tester

Error Handling Conventions

OpenLMIS would like to follow error handling best practices, this document covers the conventions we'd *like* to see followed in the various OpenLMIS components.

Java and Spring

The Java community has a long-standing debate about the proper use of Exceptions. This section attempts to be pragmatic about the use of exceptions - especially understanding the Spring community's exception handling techniques.

Exceptions in Java are broken down into two categories: those that are recovearable (checked) and those where client code can in no-way recover from the Exception (runtime). OpenLMIS *strongly* discourages the use of checked exceptions, and the following section discusses what is encouraged and why checked exceptions should be avoided.

A pattern for normal error-handling

Normal errors for the purpose of this document are things like input validation or other business logic constraints. There are a number of sources that make the claim that these types of errors are not exceptional (i.e. bad user input is to be expected normally) and therefore Java Exception's shouldn't be used. While that's generally *very* good advice, we will be using runtime exceptions (not checked exceptions) as long as they follow the best practices laid out here.

The reasoning behind this approach is two-fold:

- Runtime exceptions are used when client code *can't* recover from their use. Typically this has been used for the class of programming errors that indicate that the software encountered a completely unexpected programming error for which it should immediately terminate. We expand this definition to include user-input validation and business logic constraints for which further user-action is required. In that case the code can't recover - it has

to receive something else before it could ever proceed, and while we don't want the program to terminate, we do want the current execution to cease so that it may pop back to a Controller level component that will convert these exceptions into the relevant (non-500) HTTP response.

- Using Runtime exceptions implies that we *never* write code that catches them. We will use Spring's `@ControllerAdvice` which will catch them for us, but our code should have less "clutter" as it'll be largely devoid of routine error-validation handling.

Effectively using this pattern requires the following rules:

1. The Exception type (class) that's thrown will map one-to-one with an HTTP Status code that we want to return, and this mapping will be true across the Service. e.g. a `throw ValidationException` will always result in the HTTP Status code 400 being returned with the body containing a "nice message" (and not a stacktrace).
2. The exception thrown is a sub-type of `java.lang.RuntimeException`.
3. Client code to a method that returns RuntimeException's should never try to handle the exception. i.e. it should **not** `try {...} catch ...`
4. The only place that these RuntimeExceptions are handled is by a class annotated `@ControllerAdvice` that lives along-side all of the Controllers.
5. If the client code needs to report multiple errors (e.g. multiple issues in validating user input), then that collection of errors needs to be grouped before the exception is thrown.
6. A Handler should never be taking one of our exception types, and returning a HTTP 500 level status. This class is reserved specifically to indicate that a programming error has occurred.Reserving this directly allows for easier searching of the logs for program-crashing type of errors.
7. Handler's should log these exceptions at the DEBUG level. A lower-level such as TRACE could be used, however others such as ERROR, INFO, FATAL, WARN, etc should not.

Example

The exception

```
public class ValidationException extends RuntimeException { ... }
```

A controller which uses the exception

```
@Controller
public class WorkflowController {

  @RequestMapping(...)
  public WorkflowDraft doSomeWorkflow() {
    ...

    if (someError)
      throw new ValidationException(...);

    ...

    return new WorkflowDraft(...);
  }
}
```

The exception handler that's called by Spring should the `WorkflowController` throw `ValidationException`.

```
@ControllerAdvice
public class WorkflowExceptionHandler {
 @ExceptionHandler(ValidationException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 private Message.LocalizedMessage handleValidationException(ValidationException ve) {
   ...
   logger.debug(ve);
   return ve.getTheLocalizedMessage();
  }
}
```

Exceptions - what we don't want

Lets look at a simple example that is indicative of the sort of code we've been writing using exceptions. This example consists of a web-endpoint that returns a setting for a given key, which hands off the work to an application service layer that uses the key provided to find the given setting.

A controller (HTTP end-point) that is asked to return some setting for a given "key"

```
@RequestMapping(value = "/settings/{key}",  method = RequestMethod.GET)
public ResponseEntity<?> getByKey(@PathVariable(value = "key") String key) {
  try {
    ConfigurationSetting setting = configurationSettingService.getByKey(key);
    return new ResponseEntity<>(setting, HttpStatus.OK);
  } catch (ConfigurationSettingException ex) {
    return new ResponseEntity(HttpStatus.NOT_FOUND);
  }
}
```

The service logic that finds the key and returns it (i.e. configurationSettingService above):

```
public ConfigurationSetting getByKey(String key) throws ConfigurationSettingException
↪{
  ConfigurationSetting setting = configurationSettingRepository.findOne(key);
  if (setting == null) {
    throw new ConfigurationSettingException("Configuration setting '" + key + "' not␣
↪found");
  }
  return setting;
}
```

In this example we see that the expected end-point behavior is to either return the setting asked for and an HTTP 200 (success), or to respond with HTTP 404 - the setting was not found.

This usage of an Exception here is not what we want for a few reasons:

- The Controller directly handles the exception - it has a try-catch block. It should only handle the successful path which is when the exception isn't thrown. We should have a Handler which is `@ControllerAdvice`.
- The exception `ConfigurationSettingException` doesn't add anything - either semantically or functionally. We know that this type of error isn't that there's some type of Configuration Setting problem, but rather that something wasn't found. This could more generically and more accurately be named a `NotFoundException`. It conveys the semantics of the error and one single Handler method for the entire Spring application could handle all `NotFoundExceptions` by returning a HTTP 404.
- It's worth noting that this type of null return is handled well in Java 8's Optional. We would still throw an exception at the Controller so that the Handler could handle the error, however an author of middle-ware code

should be aware that they could use Optional instead of throwing an exception on a null immediately. This would be most useful if many errors could occur - i.e. in processing a stream.

- This code is flagged by static analysis tools with the error that this exception should be "Either log or re-throw this exception". A lazy programmer might "correct" this by logging the exception, however this would result in the log being permeated with noise from bad user input - which should be avoided.

How the API responds with validation error messages

What are Validation Error Messages?

In OpenLMIS APIs, validation errors can happen on PUT, POST, DELETE or even GET. When validation or permissions are not accepted by the API, invalid requests should respond with a helpful validation error message. This response has an HTTP response body with a simple JSON object that wraps the message. Different clients may use this message as they wish, and may display it to end-users.

The Goal: We want the APIs to respond with validation error messages in a standard way. This will allow the APIs and the UI components to all be coded and tested against one standard.

When does this pattern apply?

When does this "validation error message" pattern apply? We want to apply this pattern for all of the error situations where we return a HTTP response body with an error message. For more details about which HTTP status codes this aligns with, see the 'HTTP Status Codes' section below.

What do we return on Success?

In general, success responses should not include a validation message of the type specified here. This will eliminate the practice which was done in OpenLMIS v2, EG:

```
PUT /requisitions/75/save.json
Response: HTTP 200 OK
Body: {"success":"R&R saved successfully!"}
```

On success of a PUT or POST, the API should usually return the updated resource with a HTTP 200 OK or HTTP 201 Created response code. On DELETE, if there is nothing appropriate to return, then an empty response body is appropriate with a HTTP 204 No Content response code.

HTTP Status Codes

Success is generally a 2xx HTTP status code and we don't return validation error messages on success. Generally, validation errors are 4xx HTTP status codes (client errors). Also, we don't return these validation error messages for 5xx HTTP status codes (server or network errors). We do not address 5xx errors because OpenLMIS software does not always have control over what the stack returns for 5xx responses (those could come from NGINX or even a load balancer).

Examples below show appropriate use of HTTP 403 and 422 status codes with validation error messages. The OpenLMIS Service Style Guide includes further guidance on HTTP Status Codes that comes from Best Practices for Designing a Pragmatic RESTful API.

Example: Permissions/RBAC

The API does a lot of permission checks in case a user tries to make a request without the needed permissions. For example, a user may try to initiate a requisition at a facility where they don't have permissions. That should generate a HTTP 403 Forbidden response with a JSON body like this:

```
{
  "message" : "Action prohibited because user does not have permission at the facility
↪",
  "messageKey" : "requisition.error.prohibited.noFacilityPermission"
}
```

When creating these error validation messages, we encourage developers to avoid repeating code. It may be appropriate to write a helper class that generates these JSON validation error responses with a simple constructor.

We also don't want developers to spend lots of time authoring wordy messages. It's best to keep the messages short, clear and simple.

Translation/i18n

Message keys are used for translations. Keys should follow our Style Guide i18n Naming Conventions.

The "messageKey" is the key into a property translation file such as a .properties file maintained using Transifex or a similar tool.

The "messageKey" will be used with translation files in order to conduct translation, which we allow and support on the server-side and/or the client-side. Any OpenLMIS instance may configure translation to happen in its services or its clients.

A service will use the "messageKey" to translate responses into a different language server-side in order to respond in the language of choice for that OpenLMIS implementation instance. And/or a client/consumer may use the "messageKey" to translate responses into a language of choice.

The source code where a validation error is handled should have the "messageKey" only. The source code should not have hard-coded message strings in English or any language.

Messages with Placeholders for Translation

Placeholders allow messages to be dynamic. For example, "Action prohibited because user {0} does not have permission {1} at facility {2}".

The Transifex tool appears to support different types of placeholders, such as {0} or %s and %d. In OpenLMIS v2, the MessageService (called the Notification Service in v3) uses placeholders to make email messages translate-able. For an example, see the StatusChangeEventService.

Future: Arrays of Messages

In the future, we may extend these guidelines to support an array of multiple messages.

Future: Identifying Fields Where Validation Was Not Accepted

In the future, it may also be helpful to extend this to allow the error messages to be associated with a specific piece of data. For example, if a Requisition Validation finds that line item quantities do not add up correctly, it could provide

an error message tied to a specific product (line item) and field. Often this kind of validation may be done by the client (such as in the AngularJS UI app), and the client can immediately let the end-user know about a specific field with a validation error.

Future: Including Stack-Traces in Development Mode

In the future, it may be useful to be able to launch the entire application in a debug mode.In this mode errors returned via the API might include a stacktrace or other context normally reserved for the server log. This would be a non-default mode that developers could use to more easily develop the application.

Proposed RAML

```
schemas:
  - errorResponse: |
    { "type": "object",
      "$schema": "http://json-schema.org/draft-03/schema",
      "title": "ErrorResponse",
      "description": "Error response",
      "properties": {
        "message": { "type": "string", "required": true, "title": "error message" },
        "messageKey": { "type": "string", "required": true, "title": "key for␣
↪translations" }
      }
    }

/requisitions:
  /{id}:
    put:
      description: Save a requisition with its line items
      responses:
        403:
        422:
          body:
            application/json:
              schema: errorResponse
```

UI Conventions

See the OpenLMIS-UI Styleguide for documentation about how components look and function.

OpenLMIS-UI Styleguide

Overview

This is the styleguide for the OpenLMIS-UI, and establishes HTML patterns that should be followed to create pages and components in the OpenLMIS-UI. This documentation is generated from the source code that creates the OpenLMIS-UI.

General Rules

The following are general stylistic rules for the OpenLMIS-UI, which implementers and developers should keep in mind while crafting content.

Titles

Titles include page titles, report titles, headings within a page (H2, H3, etc), and the subject line of email notifcations. Links in the main navigation menu are generally page titles. Most other strings that appear on-screen are Labels, Buttons or others described further below.

Titles should be written so they describe a specific object and state. If there is a state that is being applied to the object in a title, the state is first in the present tense. The first letter of each word in a title should be capitalized, except for the articles of the sentence. Titles do not contain punctuation.

See APA article about title case for more guidance.

Examples Do: "Initiate Requisition" Do Not: "REQUISITION - INITIATE"

Labels

Labels are generally used in form elements to describe the content a user should input. Labels have the first letter of the first word capitalized, and should not have any punctuation such as a colon.

Labels also include table column headers and dividers for sections or categories.

Note: Colons should be added using CSS pseudo-selector, if an implementation requires labels to be formatted with a colon. As a community, we feel that less allows for easier customization.

Example Do: "First name" Do Not: "First Name:"

Buttons

Buttons should be used to refer to a user taking an action on an object, meaning there should always be a specific verb followed by a subject. Buttons have the first letter of each word capitalized and don't have any punctuation.

Example Do: "Search Facilities" Do Not: "SEARCH"

Messages

Messages represent a response from the system to a user. These strings should be written as a command, where the first word is the action that has happened. The first letter of a message is capitalized, but there is no punctuation.

Example Do: "Failed to save user profile" Do Not: "Saving user profile failed."

Confirmations

Confirmations are messages shown to the user to confirm that they actually want to take an action. These messages should address the user directly and be phrased as a single sentence.

Example Do: "Are you sure you want to submit this requisition?" Do Not: "Submitting requisition, are you sure? Please confirm."

Instructions

Instructions might be placed at the top of a form or after a confirmation to clarify the action a user is taking. These should be written as full paragraphs.

Example Do: "Authorize this requisition to send the requisition to the approval workflow." Do Not: "Authorize requisition — send to approval workflow"

OpenLMIS-UI Coding Conventions

This document describes the desired formatting to be used withing the OpenLMIS-UI repositories, many of the conventions are adapted from John Papa's Angular V1 styleguide and SMACSS by Jonathan Snook.

General

The following conventions should be applied to all sections of UI development:

- All intentation should be 4 spaces
- Legacy code should be refactored to meet coding conventions
- No thrid party libraries should be included in a OpenLMIS-UI repository

File Structure

All file types should be organized together within the `src` directory according to functionality, not file type — the goal is to keep related files together.

Use the following conventions:

- File names are lowercase and dash-seperated
- Files in a directory should be as flat as possible (avoid sub-directories)
- If there are more than 12 files in a directory, try to divide files into subdirectories based on functional area

Each file type section below has specifics on their naming conventions

HTML Markup Guidelines

Less markup is better markup, and semantic markup is the best.

This means we want to avoid creating layout specific markup that defines elements such as columns or icons. Non-semantic markup can be replicated by using CSS to create columns or icons. In some cases a layout might not be possible without CSS styles that are not supported across all of our supported browsers, which is perfectly acceptiable.

Here is a common pattern for HTML that you will see used in frameworks like Twitter's Bootstrap (which we also use)

```
<li class="row">
    <div class="col-md-9">
        Item Name
    </div>
    <div class="col-md-3">
        <a href="#" class="btn btn-primary btn-block">
            <i class="icon icon-trash"></i>
```

```
            Delete
        </a>
    </div>
</li>
<div class="clearfix"></div>
```

The above markup should be simplified to:

```
<li>
    Item Name
    <button class="trash">Delete</button>
</li>
```

This gives us simpler markup, that could be restyled and reused depending on the context that the HTML section is inserted into. We can recreate the styles applied to the markup with CSS such as:

- A ::before pseudo class to display an icon in the button
- Using float and width properties to correctly display the button
- A ::after pseudo class can replace any 'clearfix' element (which shouldn't exist in our code)

See the UI-Styleguide for examples of how specific elements and components should should be constructed and used.

Naming Convention

In general we follow the John-Papa naming conventions, later sections go into specifics about how to name a specific file type, while this section focuses on general naming and file structure.

Generally, all file names should use the following format `specific-name.file-type.ext` where:

- `specific-name` is a dash-separated name for specific file-type
- `file-type` is the type of object that is being added (ie 'controller', 'service', or 'layout')
- `ext` is the extention of the file (ie '.js', '.scss')

Folder structure should aim to follow the LIFT principal as closely as possible, with a couple extra notes:

- There should only be one *.module.js file per directory hiearchy
- Only consider creating a sub-directory if file names are long and repatitive, such that a sub-directory would improve meaning

SASS & CSS Formatting Guidelines

General SASS and CSS conventions:

- Only enter color values in a variables file
- Only enter pixel or point values in a variables file
- Variable names should be lowercase and use dashes instead of spaces (ie: *$sample-variable*)
- Avoid class names in favor of child element selectors where ever possible
- Files should be less than 200 lines long
- CSS class names should be lowercase and use dashes instead of spaces

SMACSS

The CSS styles should reflect the SMACSS CSS methodology, which has 3 main sections — base, layout, and module. SMACSS has other sections and tennants, which are useful, but are not reflected in the OpenLMIS-UI coding conventions.

Base

CSS styles applied directly to elements to create styles that are the same throughout the application.

Layout

CSS styles that are related primarly to layout in a page — think position and margin, not color and padding — these styles should never be mixed with base styles (responsive CSS should only be implemented in layout).

Module

This is a css class that will modify base and layout styles for an element and it's sub-elements.

SASS File-Types

Since SASS pre-processes CSS, there are 3 SCSS file types to be aware of which are processed in a specific order to make sure the build process works correctly.

Variables

A variable file is either named 'variables.scss' or matches '*.variables.scss'

Varriables files are the first loaded file type and include any variables that will be used through out the application — *There should be as few of these files as possible*.

The contents of a varriables file should only include SASS variables, and output no CSS at anypoint.

There is no assumed order in which varriables files will be included, which means:

- Varriable files shouldn't have overlapping varriables
- Implement SASS's variable default (!default)

Mixins

A mixin file matches the following pattern *.mixin.scss

Mixins in SASS are reusable functions, which are loaded second in our build process so they can use global variables and be used in any other SCSS file.

There should only be one mixin per file, and the file name should match the function's name, ie: 'simple-function.mixin.scss'

All Other SCSS and CSS Files

All files that match '*.scss' or* '.css' are loaded at the same time in the build process. This means that no single file can easily overwrite another files CSS styles unless the style is more specific or uses `!imporant` — This creates the following conventions:

- Keep CSS selectors as general as possible (to allow others to be more specific)
- Avoid using !important

To keep file sizes small, consider breaking up files according to SMACSS guidelines by adding the type of classes in the file before .scss or .css (ie: `navigation.layout.scss`)

Javascript Formatting Guidelines

General conventions:

- All code should be within an immedately invoked scope
- *ONLY ONE OBJECT PER FILE*
- Variable and function names should be written in camelCase
- All Angular object names should be written in CamelCase

Documentation

To document the OpenLMIS-UI, we are using ngDocs built with grunt-ngdocs. See individual object descriptions for specifics and examples of how to document that object type.

General rules

- any object's exposed methods or variables must be documented with ngDoc
- @ngdoc annotation specifies the type of thing being documented
- as 'Type' in documentation we should use:
- Promise
- Number
- String
- Boolean
- Object
- Event
- Array
- Scope
- in some cases is allowed to use other types i.e. class names like Requisition
- all description blocks should be sentence based, all of sentences should start with uppercase letter and end with '.'
- before and after description block (if there is more content) there should be an empty line

- all docs should be right above the declaration of method/property/component
- when writing param/return section please keep all parts(type, parameter name, description) start at the same column as it is shown in method/property examples below
- please keep the order of all parameters as it is in examples below

General Object Documentation

Regardless of the actual component's type, it should have '@ngdoc service' annotation at the start, unless the specific object documentation says otherwise. There are three annotations that must be present:

- ngdoc definition
- component name
- and description

```
/**
 * @ngdoc service
 * @name module-name.componentName
 *
 * @description
 * Component description.
 */
```

Methods

Methods for all components should have parameters like in the following example:

```
/**
 * @ngdoc method
 * @methodOf module-name.componentName
 * @name methodName
 *
 * @description
 * Method description.
 *
 * @param  {Type} paramsName1 param1 description
 * @param  {Type} paramsName2 (optional) param2 description
 * @return {Type}             returned object description
 */
```

Parameters should only be present when method takes any. The same rule applies to return annotation. If the parameter is not required by method, it should have "(optional)" prefix in the description.

Properties

Properties should be documented in components when they are exposed, i.e. controllers properties declared in 'vm'. Properties should have parameters like in the following example:

```
/**
 * @ngdoc property
 * @propertyOf module-name.componentName
 * @name propertyName
```

```
 * @type {Type}
 *
 * @description
 * Property description.
 */
```

Unit Testing Guidelines

A unit tests has 3 goals that it should accomplish to test a javascript object:

- Checks success, error, and edge cases
- Tests as few objects as possible
- Demonstrates how an object should be used

With those 3 goals in mind, its important to realize that the variety of AngularJS object types means that the same approact won't work for each and every object. Since the OpenLMIS-UI coding conventions layout patterns for different types of AngularJS objects, it's also possible to illustrate how to unit test objects that follow those conventions.

Check out AngularJS's unit testing guide, its well written and many of out tests follow their styles.

Here are some general rules to keep in mind while writing any unit tests:

- Keep beforeEach statements short and to the point, which will help other's read your statements
- Understand how to use Spies in Jasmine, they can help isolate objects and provide test cases

Angular V1 Object Guidelines

AngularJS has many different object types — here are the following types the OpenLMIS-UI primarily uses. If there is a need for object types not documented, please refer to the John Papa Angular V1 styleguide.

Modules

Modules in angular should describe and bind together a small unit of functionality. The OpenLMIS-UI build process should construct larger module units from theses small units.

Documentation

Docs for modules must contain the module name and description. This should be thought of as an overview for the other obects within the module, and where appropriate gives an overview of how the modules fit together.

```
/**
 * @module module-name
 *
 * @description
 * Some module description.
 */
```

Replaced Values

@@ should set own default values

Constants

Constants are Javascript variables that won't change but need to be resued between multiple objects within an Angular module. Using constants is important because it becomes possible to track an objects dependencies, rather than use variables set on the global scope.

It's also useful to wrap 3rd party objects and libraries (like jQuery or bootbox) as an Angular constant. This is useful because the dependency is declared on the object. Another useful feature is that if the library or object isn't included, Angualr will throw a single verbose error message.

Add rule about when its ok to add a group of constants – if a grouping of values, use a plural name

Conventions:

- All constant variable names should be upper case and use underscores instead of spaces (ie VARIABLE_NAME)
- If a constant is only relivant to a single Angular object, set it as a variable inside the scope, not as an Angular constant
- If the constant value needs to change depending on build variables, format the value like @@VARIABLE_VALUE, and which should be replaced by the grunt build process if there is a matching value
- Wrap 3rd party services as constants, if are not already registered with Angular

Service

John Papa refers to services as Singletons, which means they should only be used for application information that has a single instance. Examples of this would include the current user, the application's connection state, or the current library of localization messages.

Conventions

- Services should always return an object
- Services shouldn't have their state changed through properties, only method calls

Naming Convention

nameOfService*Service*

Always lowercase camelCase the name of the object. Append 'Service' to the end of the service name so developers will know the object is a service, and changes will be persisted to other controllers.

Unit Testing Conventions

- Keep $httpBackend mock statements close to the specific places they are used (unless the statement is reusable)
- Use Jasmine's spyOn method to mock the methods of other objects that are used
- In some cases mocking an entire AngularJS Service, or a constant, will be required. This is possible by using AngularJS's $provide object within a beforeEach block. This would look like

```
beforeEach(module($provide){
    // mock out a tape recorder service, which is used else where
    tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);

    // overwrite an existing service
    $provide.service('TapeRecorderService', function(){
        return tape;
    });
});
```

Factory

Factories should be the most used Angualr object type in any application. John Papa insists that factories serve a single purpose, and should be extended by variabled they are called with.

This means that factories should generally return a function that will return an object or set of objects that can be manipulated. It is common for a factory to include methods for interacting with a server, but this isn't nessicarry.

Should be used with UI-Router resolves, and get additional arguments

Naming Convention

specificName*Factory*

Factories should always be named lowercase camelCase. To avoid confussion between created objects and factories, all factories should have the word'Factory' appended to the end (this disagrees with John-Papa style).

Example

```
angular.module('openlmis-sample')
    .factory('sampleFactory', sample);

sample.$inject = [];
function sample(){
    var savedContext;

    return {
        method: method,
        otherMethod: otherMethod
    }
}
```

Unit Testing Conventions Test a factory much like you would test a service, except be sure to:

- Declare a new factory at the start of every test
- Exercise the producted object, not just the callback function

Javascript Class

Pure javascript classes should only be used to ease the manipulation of data, but unlike factories, these object shouldn't create HTTP connections, and only focus on a single object.

Javascript classes should be injected and used within factories and *some services* services that have complex logic. Modules should be able to extend javascript classes by prototypical inheritance.

Helps with code reusability

Requisition/LineItem is good example

Naming Conventions

SampleName

Classes should be uppercase CamelCased, which represents that they are a class and need to be instantiated like an object (ie `new SampleName()`).

Controller

Controllers are all about connecting data and logic from Factories and Services to HTML Views. An ideal controller won't do much more than this, and will be as 'thin' as possible.

Controllers are typically specific in context, so as a rule controllers should never be reused. A controller can be linked to a HTML form, which might be reused in multiple contexts — but that controller most likely wouldn't be applicable in other places.

It is also worth noting that John Papa insists that controllers don't directly manipulate properties in $scope, but rather the ControllerAs syntax should be used which injects the controller into a HTML block's context. The main rationale is that it makes the $scope variables less cluttered, and makes the controller more testable as an object.

Conventions

- Should be only object changing application $state
- Is used in a single context
- Don't use the $scope variable EVER
- Use ControllerAs syntax
- Don't $watch variables, use on-change or refactor to use a directive to watch values

Unit Testing

- Set all items that would be required from a route when the Controller is instantiated
- Mock any services used by the controller

Documentation

The only difference between controllers and other components is the '.controller:' part in the @name annotation. It makes controller documentation appear in controllers section. Be sure to document the methods and properties that the controller exposes.

```
/**
 * @ngdoc service
 * @name module-name.controller:controllerName
 *
 * @description
 * Controller description.
 *
 */
```

Routes

Routing logic is defined by UI-Router, where a URL path is typically paired with an HTML View and Controller.

Use a factory where possible to keep resolve statements small and testable

General Conventions

- The UI-Router resolve properties are used to ease loading on router
- Routes should define their own views, if their layout is more complicated than a single section

HTTP Interceptor

HTTP Interceptors are technically factories that have been configured to 'intercept' certain types of requests in Angular and modify their behavior. This is recommended because other Angular objects can use consistent Angular objects, reducing the need to write code that is specialized for our own framework.

Keep all objects in a single file - so its easier to understand the actions that are being taken

The Angular guide to writting HTTP Interceptors is here

General Conventions

- Write interceptors so they only chanage a request on certain conditions, so other unit tests don't have to be modified for the interceptors conditions
- Don't include HTTP Interceptors in openlmis-core, as the interceptor might be injected into all other unit tests — which could break everything

Unit Testing Conventions

The goal when unit testing an interceptor is to not only test input and output transformation functions, but to also make sure the interceptor is called at an appropriate time.

Directive

Directives are pieces of HTML markup that have been extended to do a certain function. *This is the only place where it is reasonable to manipulate the DOM.*

Make disticntion between directive and component – components use E tag and isolate scope, directive use C and never isolate scope

Conventions

- Restrict directives to only elements or attributes
- Don't use an isolated scope unless you absolutely have to
- If the directive needs extenal information, use a controller — don't manipulate data in a link function

Unit Testing

The bit secrect when unit testing a directive is to make sure to use the $compile function to return an element that is extended with jQuery. Once you have this object you will be able to interact with the directive by clicking, hovering, or triggering other DOM events.

```
describe('SampleDirective', function(){
    it('gets compiled and shows the selected item name', function($compile,
↪$rootScope){
        var scope = $rootScope.$new();
        scope['item'] = {
            name: "Sample Title"
        };
        var element = $compile("<sample-directive selected='item'></sample-directive>
↪")(scope);

        expect(element.text()).toBe("Sample Title");
    });
    it('responds to being clicked', function($compile, $rootScope){
        var element = $compile("<sample-directive selected='item'></sample-directive>
↪")($rootScope.$new());

        // check before the action
        expect(element.text()).toBe("No Title");

        element.click();
        // check to see the results of the action
        // this could also be looking at a spy to see what the values are
        expect(element.text()).toBe("I was clicked");
    });
});
```

Documentation

Directive docs should have well described '@example' section.

Directive docs should always have '@restrict' annotation that takes as a value one of: A, E, C, M or any combination of those. In order to make directive docs appear in directives section there needs to be '.directive:' part in @name annotation.

```
/**
 * @ngdoc directive
 * @restrict A
 * @name module-name.directive:directiveName
 *
 * @description
 * Directive description.
 *
```

```
 * @example
 * Short description of how to use it.
 * ```
 *   <div directiveName></div>
 * ```
 * Now you can show how the markup will look like after applying directive code.
 * ```
 * <div directiveName>
 *     <div>something</div>
 * </div>
 * ```
 */
```

Modal

A modal object isn't a 'native Angular object' — it is a service or factory that displays a modal window. This is done for convience and because it allows modal windows to not be declared in html files — and be used more easily by controllers (or even services, if appropriate).

Use Javascript class

Conventions

Unit Tests

When creating a unit test for a modal service, the unit tests should focus on event driven logic and avoid testing functionality that is tied to the DOM. Since we are using Bootbox to manage the creation of modal elements, we can mock Bootbox and trust the Bootbox will successfully interact with the DOM.

```
// Imagine testing a modal that will show an alert, and when closed will resolve a␣
↪promise.
describe('SampleModal', function(){

    // Instead of doing a beforeEach (recommended), this example directly injects␣
↪dependencies
    it('when closed will resolve promise', function($rootScope, SampleModal, bootbox){

        // Pull out the callback that will be passed to bootbox when the window closes
        var closeCallback;
        spyOn(bootbox, 'alert').andCallFake(function(argumentObject){
            closeCallback = argumentObject.callback;

            // Bootbox is supposed to return a jQuery element, which we will mock␣
↪with an object
            return {};
        });

        // make a spy to track if the promise works
        var promiseSpy = createSpy();

        // Make the modal, and save the promise...
        var promise = SampleModal().then(promiseSpy);

        // If we check the promiseSpy immedately, it shouldn't have been
```

```
        // called because the closeCallback wasn't called...
        expect(promiseSpy).not.toHaveBeenCalled();

        // Call closeCallback, which is out mocked version of clicking the
        // "ok" button on the alert modal
        closeCallback();
        expect(promiseSpy).toHaveBeenCalled();
    });
});
```

Filters

[Stub]

Documentation

Filter docs should follow the pattern from example below:

```
/**
 * @ngdoc filter
 * @name module-name.filter:filterName
 *
 * @description
 * Filter description.
 *
 * @param   {Type} input     input description
 * @param   {Type} parameter parameter description
 * @return  {Type}           returned value description
 *
 * @example
 * You could have short description of what example is about etc.
 * ```
 * <div>{{valueToBeFiltered | filterName:parameter}}</div>
 * ```
 */
```

It is a good practice to add example block at the end to make clear how to use it. As for parameters the first one should be describing input of the filter. Please remember of '.filter:' part. It will make sure that this one will appear in filters section.

HTML Views

Angular allows HTML files to have variables and simple logic evaluated within the markup.

A controller that has the same name will be the reference to vm, if the controller is different, don't call it vm

General Conventions

- If there is logic that is more complicated than a single if statement, move that logic to a controller
- Use filters to format variable output — don't format variables in a controller

Memory Leaks

This one is a bit tricky. It's fairly hard to create a memory leak in AngularJS unless you're mixing it with other external libraries that are not based on AngularJS(especially jQuery). Still, there are some things you need to remember while working with it, this article provides some general insight on how to find, fix and avoid memory leaks, for more detailed info I would suggest reading this article(it's awesome!).

Finding memory leaks

I won't lie, finding out if your application has some memory leaks is annoying, and localizing those leaks is even more annoying and can take a lot of time. Google Chrome devtools is incredible tool for doing this. All you need to do is:

1. open you application
2. go to the section you want to check for memory leaks
3. execute the workflow you want to check for memory leaks so any service or cached data won't be shown on the heap snapshot
4. open devtools
5. go to the Profiles tab
6. select Take Heap Snapshot
7. take a snapshot
8. execute the workflow
9. take a snapshot again
10. go to a different state
11. take a snapshot again
12. select the last snapshot
13. now click on the All objects select and choose Objects allocated between Snapshot 1 and Snapshot 2

This will show you the list of all objects, elements and so on, that were created during the workflow and are still residing in the memory. That was the easy part. Now we need to analyze the data we have and this might be quite tricky. We can click on object to see what dependency is retaining them. There is some color coding here that can be useful to you - red for detached elements and yellow for actual code references which you can inspect and see. It takes some time and experience to understand what's going here but it gets easier and easier as you go.

Anti-patterns

Here are some anti-pattern that you should avoid and how to fix them.

Event handlers using scope

Let's look at the following example. We have a simple directive that binds an on click action to the element.

```
(function() {

    'use strict';

    angular
```

```
        .module('some-module')
        .directive('someDirective', someDirective);

    function someDirective() {
        var directive = {
            link: link
        };
        return directive;

        function link(scope, element) {

            element.on('click', onClick);

            function onClick() {
                scope.someFlag = true;
            }
        }
    }

})();
```

The problem with this link function is that we've created a closure with context which retains the context, the scope and "then basically everything in the universe" until we unregister the handler from the element. That's right, even after the element is removed from the DOM it will still reside in the memory retained by the closure unless unregister the handler. To do this we need to add a handler for '$destroy' event to the scope object and then unregister the handler from the element. Here's an example how to do it.

```
(function() {

    'use strict';

    angular
        .module('some-module')
        .directive('someDirective', someDirective);

    function someDirective() {
        var directive = {
            link: link
        };
        return directive;

        function link(scope, element) {

            element.on('click', onClick);

            scope.$on('$destroy', function() {

                //this will unregister the this single handler
                element.off('click', onClick);

                //this will unregister all the handlers
                element.off();
            });


            function onClick() {
                scope.someFlag = true;
            }
```

```
        }
    }

})();
```

Improper use of the $rootScope.$watch method

$rootScope.$watch can be a powerful tool, but it also requires some experience to use right. It allows the developers to create watchers that live through the whole application life and are only removed when they are explicitly said to unregister or when the application is closed, which may result in a huge memory leaks. Here are some tips on how to use them.

- Use $scope.$watch when possible! If you're using a watcher in a directive, it will have access to the scope object, add the watcher to it! This way we take advantage of AngularJS automatic watcher unregistration when the scope is deleted.
- Avoid using $rootScope.$watch in factories. Don't use it in factories unless you're completely sure what you're doing. Remember to unregister it when it is no longer needed! This takes us to the next bullet point.
- Use them in Services. Watching for current locale can be great example of that. We're using it with service, which is a singleton - it is only created once during application lifetime - and we want to watch for the current locale all the time we rather won't want to stop at any point.
- Unregister it if it is no longer needed. If you're sure you won't be needing that watcher any longer simply unregister it! Here's an example

```
var unregisterWatcher = $rootScope.$watch('someVariable', someMethod);
unregisterWatcher();
```

Using callback functions

Using callback isn't the safest idea either as it can cause some function retention. AngularJS gives us awesome tool to bypass that - promises. They basically gives us the same behavior and are retention-risk free!

Patterns

See JS Documentation for more details

List View Pattern

Pagination Patterns

STUB

Sorting Pattern

STUB

Offline Pattern

Deployment

Deployment is done currently through Docker and Docker Compose. A living example of deployment scripts and documentation that the OpenLMIS product uses to deploy demo and CD environments is available in the openlmis-deployment repository. Documentation from that repository is listed below:

Recommended Deployment Topology

OpenLMIS uses and therefore recommends that most deployments utilize Amazon Web Services (AWS). However OpenLMIS is in no way tied to only being deployed on AWS.

The basic ingredients of an OpenLMIS deployment are:

- a domain name to reach the installation at (e.g. test.openlmis.org)
- a SSL certificate to make the communication to OpenLMIS secure over the web
- a computer/instance/etc that can run Docker Machine (as well as Compose, etc) with enough bandwidth, processing power, memory and storage to run many (6+) Services and associated utilities
- a computer/instance/etc that can run PostgreSQL for those Services
- credentials with an SMTP server to send emails

In AWS this would look like:

- A DNS provider for your domain name (e.g. test.openlmis.org). This could be Route 53, however currently OpenLMIS deployments do not utilize this service.
- A SSL certificate from AWS Certificate Manager
- A ELB that can route to/from the OpenLMIS instance and serve the ACM SSL certificate (this becomes more useful when running out of Elastic IPs)
- an EC2 Instance (m4.large - 2vCPU, 8GiB memory, 30GB EBS store)
- an RDS Instance (you could start with the smallest one, and then upgrade based on need)
- a VPC for your EC2 and RDS instances, with appropriate security group - SSH, HTTP, HTTPs, Postgres (limit source to Security Group) at minimum.
- Amazon SES with either the domain (w/DKIM) verified or a specific from-address

For more information on setting this up, see the Provisioning section and also follow the link for backing up/restoring RDS. For more information on how to configure your RDS please visit RDS configuration page.

How to provision a single Docker host in AWS

If case the deployment target is one single host, then swarm is not needed.

In that case, refer to Provision-swarm-With-Elastic-ip.md for step 1, 2, 3, 5. Omit step 4, that should be sufficient to provision a single host deployment environment.

Note: choose **ubuntu** instead of amazon linux distribution. Even though this single host won't be running a swarm, ubuntu is still preferred over amazon linux distribution. Because docker-machine does not support provisioning amazon linux distribution. However, you can manually provision the single host, but then making that host remotely accessible would be tricky and involves a lot of manual steps.

Database

If deploying OpenLMIS with the included Docker Container for Postgres, then no further steps are needed. However this setup is recommended only for development / testing environments and not recommended for production installs.

Test and UAT environments in this repository demonstrate that Postgres could be installed outside of Docker and OpenLMIS services may be pointed to that Postgres server. Test and UAT both use Amazon's RDS service to help manage production-grade database services such as automated patch release updates, rolling backups, snapshots, etc.

Some notes on provisioning an RDS instance for OpenLMIS:

- Test and UAT are both capable of running on economical RDS instances: db.t2.micro
- When choosing a small RDS instance, the max number of connections are set based on an *ideal* number from RDS. OpenLMIS services tend toward using about 10 DB connections per service. Therefore Tests and UAT instances use a Parameter Group named Max Connections that increase this limit to 100. Larger, more expensive, instances likely won't have this limitation.
- RDS instances are in a private VPC and in the same availability zone as their EC2 instance and it's ELB. The security group used should be the same as used for the EC2 instance, though it should limit incoming PostgreSQL connections to only those from the security group.
- Don't forget to update the .env file used to deploy OpenLMIS with the correct Host, username and password settings.

How to provision a docker swarm for deployment in AWS (With ELB)

1. Network setup

Create a VPC for the new swarm cluster.

In it, there should be 2 subnets created(which is needed by AWS ELB later).

Ensure the subnets will assign public ip to EC2 instances automatically, so it's easier to ssh into them later.

2. Create EC2 instances

This step is similar to creating EC2 instances for any other type of purpose.

When creating those instances, make sure to select the VPC created in the previous step.

Mentally mark one of the instances as swarm manager, the rest of them will be regular nodes.

Note: choose **ubuntu** instead of amazon linux distribution. The amazon linux distribution has problems with docker 1.12, the version that has built in support for docker swarm. 1.12 is not available in amazon linux RPM yet. And it also lacks support for aufs, which is recommended by docker.

Make sure to open port 2376 insecurity group, this is the default port that docker-machine uses to provision.

3. Add ssh public key to the newly created EC2 instances

In order to access the EC2 instances, the public key of **the machine from which the provisioning will happen** need to be added to the target machine.

This is done by ssh into the EC2 instances, and then edit `[User Home Dir]/.ssh/authorized_keys` file to add your public key into it.

This will be needed by the `docker-machine create` command later.

4. Create ELB

The reason to create ELB is that AWS has a limit on how many elastic ips each account could have, the default is 5, which could be easily used up.

So in order for the swarm to be available via a constant address, an ELB is created to provide that constant url.

This is also why in the first step, there need to be 2 subnets, it's required by ELB.

When creating the ELB, make sure **TCP port 22 and 2376** are forwarded to the target EC2 instance. 22 is for ssh, 2376 is for docker remote communication. And also, make sure to choose classic ELB instead of the new one. The classic one allows TCP forwarding, the new one only supports http and https.

5. Enable health check

ELB only forwards to a target machine if the target is considered "healthy". And ELB determines the health of a target by pinging it.

So, in the EC2 instance chosen as the swarm manager, start apache2 service at port 80(or any other port you may prefer). Then in ELB settings, set it to ping that port.

For OpenLMIS the Nginx container starts itself automatically after the system is rebooted, so this ensures that **ELB will start forwarding immediately if the instance reboots**.

6. Provision all EC2 instances

Use this command:

```
docker-machine create --driver generic --generic-ip-address=[ELB Url]
--generic-ssh-key ~/.ssh/id_rsa --generic-ssh-user ubuntu name1
```

to provision the swarm manager.

Note: the –driver flag has support for AWS. But the intention to explicitly ***not*** use it is to make sure this provision step could apply to other host environments as well, not just AWS hosted machines.

The –generic-ip-address flag needs to be followed by the ip of ec2 instance(for the swarm manager, it should be the ELB Url).

The –generic-ssh-key flag needs to be followed private key, whose public key pair should have already been added in step 2.

The –generic-ssh-user flag needs to be followed by the user name, in the case of Ubuntu EC2 instances, the default user name is ubuntu.

Lastly, supply a **name** for the docker machine.

Do this **for all the EC2 instances**, to make sure docker is installed on all of them. (When doing this for the none manager nodes, the –generic-ip-address flag should be followed by their public ip that was automatically assigned, since ELB only forwards traffic to the manager node.)

7. Start swarm

Choose one of the EC2 instances as the swarm manager by:

`eval $(docker-machine env [name of the chosen one])` (the name in the [] should be one of the names used in the previous step)

Now your local docker command is pointing at the remote docker daemon, run:

```
docker swarm init
```

Then follow its console output to join the rest of the EC2 instances into the swarm. (it could be done by switching docker-machine env, or by using the -H flag of docker, the former is easier)

Since all the swarm node are in the same VPC, they can talk to each other by private ips which are static inside the VPC. **The swarm will regroup it self and maintain the manager-regular node structure even after EC2 instances are rebooted.**

8. Allow Jenkins to access swarm manager

In order for Jenkins to continuously deploy to the swarm, it needs access to the swarm manager.

In step 3, when the swarm manager EC2 instance was being provisioned. The docker-machine created some certificate files behind the scene.

Those files should be in the machine that the provision command was issued(not the machine that was being provisioned), under:

```
[User Home Dir]/.docker/machine/machines/[name of the swarm manager]
```

Those files need to be copied to jenkins.

In a Jenkins deployment job, **at the start of its build script**, add:

```
export DOCKER_TLS_VERIFY="1"
```

```
export DOCKER_HOST="tcp://[ELB Url that forwards to Swarm manager]"
```

```
export DOCKER_CERT_PATH="[path to the dir that contains certs]"
```

This will make following docker commands use the remote daemon, not the local one.

Now, Jenkins should be able to access and deploy to the swarm.

Node: Jenkins would only need access to the swarm manager, the other nodes are managed by the swarm manager. Jenkins does not need direct access to them.

How to provision a docker swarm for deployment in AWS (With Elastic IP)

1. Create EC2 instances

This step is similar to creating EC2 instances for any other type of purpose.

Mentally mark one of the instances as swarm manager, the rest of them will be regular nodes. Assign an elastic ip to the manager node.

Note: choose **ubuntu** instead of amazon linux distribution. The amazon linux distribution has problems with docker 1.12, the version that has built in support for docker swarm. 1.12 is not available in amazon linux RPM yet. And it also lacks support for aufs, which is recommended by docker.

Make sure to open port 2376, this is the default port that docker-machine uses to provision. And make sure they have auto assigned public ip(not elastic ip) so you can ssh into them.

2. Add ssh public key to the newly created EC2 instances

In order to access the EC2 instances, the public key of **the machine from which the provisioning will happen** need to be added to the target machine.

This is by ssh into the EC2 instances, and then edit `[User Home Dir]/.ssh/authorized_keys` file to add your public key into it.

3. Provision all EC2 instances

With this command:

```
docker-machine create --driver generic --generic-ip-address=*.*.*.*
--generic-ssh-key ~/.ssh/id_rsa --generic-ssh-user ubuntu name1
```

Note: the –driver flag has support for AWS. But the intention to explicitly ***not*** use it is to make sure this provision guide could apply to any host machine, not just AWS hosted machines.

The –generic-ip-address flag needs to be followed by the ip of ec2 instance. For the manager node, use the elastic ip, for the others, use the temp ips assigned by aws.

The –generic-ssh-key flag needs to be followed private key, whose public key pair should have already been added in step 2.

The –generic-ssh-user flag needs to be followed by the user name, in the case of Ubuntu EC2 instances, the default user name is ubuntu.

Lately, supply a **name** for the docker machine.

Do this **for all the EC2 instances**, to make sure docker is installed on all of them.

4. Start swarm

Choose one of the EC2 instances as the swarm manager by:

`eval $(docker-machine env [name of the chosen one])` (the name in the [] should be one of the names used in the previous step)

Now your local docker command is pointing at the remote docker daemon, run:

```
docker swarm init
```

Then follow its console output to join the rest of the EC2 instances into the swarm. (it could be done by switching docker-machine env, or by using the -H flag of docker, the former is easier)

5. Allow Jenkins to access swarm manager

In order for Jenkins to continuously deploy to the swarm, it needs access to the swarm manager.

In step 3, when the swarm manager EC2 instance was being provisioned. The docker-machine created some certificate files behind the scene.

Those files should be in the machine that the provision command was issued(not the machine that was being provisioned), under:

```
[User Home Dir]/.docker/machine/machines/[name of the swarm manager]
```

Those files need to be copied to jenkins(if the provision was done on Jenkins, then there is no need to copy).

In a Jenkins deployment job, **at the start of its build script**, add:

```
export DOCKER_TLS_VERIFY="1"
```

```
export DOCKER_HOST="tcp://[ip of the swarm manager]"
```

```
export DOCKER_CERT_PATH="[path to the dir that contains certs]"
```

This will make following docker commands use the remote daemon, not the local one.

Now, Jenkins should be able to access and deploy to the swarm.

Node: Jenkins would only need access to the swarm manager, the other nodes are managed by the swarm manager. Jenkins does not need direct access to them.

Deployment Environments

Scripts in this directory are meant to be ran in Jenkins.

Overview

- `shared/` contains scripts for the Jenkins job(s):
  - `init_env.sh` is run in Jenkins to copy the docker environment files (has secure credentials) from `JENKINS_HOME/credentials/` to the current job's workspace
  - `pull_images.sh` always pulls/refreshes the infrastructure images (e.g. db, logs, etc), and then at the end will pull the image for the service that the Jenkins job is attempting to deploy (e.g. requistion, auth, referencedata, etc).
  - `restart.sh` is paramartized by Jenkins to either `keep` or `wipe` volumes (e.g. database and logging volumes). When run this brings the deployed reference distribution down, and then back up. After it's brought up, the `nginx.tmpl` file is copied directly into the running nginx container just started.
  - `nginx.tmpl` is the override of the nginx template for docker and proxying - this is a copy from openlmis-ref-distro. See `restart.sh` for how it's used.
- `test_env` has a compose file which is the Reference distribution, and a script for Jenkins to kick everything off.
- `uat_env` has a compose file which is the Reference distribution, and a script for Jenkins to kick everything off.
- `demo_env` has a compose file which is the latest stable version of the Reference distrubution, and a script for Jenkins to kick everything off.

Local Usage

These scripts **won't** work out of the box in a dev's local machine, to make them work, you need a few files that are present in Jenkins but not in your local clone of this repo:

1. The .env file

   This file is present in Jenkins. It is copied to the workspace of a deployment job(either Jenkins slave or master) every time that job is ran.

   This file is **not** included in this repo because the db credentials could be different for different deployment environments. The default .env file that is used during development and CI is open in github, making it not suitable for deployment purposes.

2. The cert files for remotely controlling docker daemon deployment target

   These files should not be included in this public repo for obvious reasons.

   Similar to the .env file, they are also present in Jenkins and copied to a deployment job's workspace(either Jenkins slave or master) every time it is ran.

To get these files, you need to be able to ssh to the Jenkin's host instance.

It's **not** recommended that you connect to the remote deployment environments, however if you have to:

1. pull the remote cert files to a local directory. They are currently located under `JENKINS_HOME/credentials/` in the `Host` directories. `JENKINS_HOME` would currently be `/var/lib/jenkins`.
2. With the above cert files, you could control the remote docker machine by copying the certs to a local `certs/` directory, and then running the following in your shell:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://<path-to-elb-of-docker-host>:2376"
export DOCKER_CERT_PATH="${PWD}/certs"
```

e.g. a current elb path for test is `elb-test-env-swarm-683069932.us-east-1.elb.amazonaws.com`

After this, running docker commands in your shell will be ran against the remote machine. e.g. docker inspect, logs, etc

How to backup persisted data?

if using ref distro's included db container

1. ssh into the docker host that you want, either test env or UAT env. Or use the technique above to connect your docker client to the remote host as needed
2. run this command

   ```
   docker exec -t [PostgresContainerName] /usr/lib/postgresql/9.4/bin/
   pg_dumpall -c -U [DBUserName] > [DumpFileName].sql
   ```

   PostgresContainerName is usually testenv_db_1 or uatenv_db1, you can use `docker ps` to find out. **DBUserName** is the one that was specified in the .env file, it's usually just "postgres". **DumpFileName** is the file name where you want the back up to be stored **in the host machine**.

using Amazon's RDS

RDS provides a number of desirable features that are more ideal for production environments, including automated backups. To backup and restore the OpenLMIS database when using RDS, follow Amazon's documentation: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_CommonTasks.BackupRestore.html

Versioning and Releasing

Micro-Services are Versioned Independently

OpenLMIS version 3 introduced a micro-services architecture where each component is versioned and released independently. In addition, all the components are packaged together into a Reference Distribution. When we refer to OpenLMIS 3.X.Y, we are talking about a release of the Reference Distribution, called the ref-distro in GitHub. The components inside ref-distro 3.X.Y have their own separate version numbers which are listed on the Release Notes.

The components are each semantically versioned, while the ref-distro has "milestone" releases that are conducted roughly quarterly (every 3 months we release 3.2, 3.3, etc). Each ref-distro release includes specific versions of the other components, both service components and UI components.

Where We Publish Releases

All OpenLMIS source code is available on GitHub, and the components have separate repositories. Releases are tagged on GitHub for all components as well as the ref-distro. Releases of some components, such as the service components and UI components, are also published to Docker Hub as versioned docker images. In addition, we publish releases of the service utility library to Maven.

Release Numbering

Version 3 components follow the Semantic Versioning standard:

- **Patch** releases with bug fixes, small changes and security patches will come out on an as-needed schedule (1.0.1, 1.0.2, etc). Compatibility with past releases under the Major.Minor is expected.
- **Minor** releases with new functionality will be backwards-compatible (1.1, 1.2, 1.3, etc). Compatibility with past releases under the same Major number is expected.
- **Major** releases would be for non-backwards-compatible API changes. When a new major version of a component is included in a Reference Distribution release, the Release Notes will document any migration or upgrade issues.

The Version 3 Reference Distribution follows a milestone release schedule with quarterly releases. Release Notes for each ref-distro release will include the version numbers of each component included in the distribution. If specific components have moved by a Minor or Major version number, the Release Notes will describe the changes (such as new features or any non-backwards-compatible API changes or migration issues).

Version 2 also followed the semantic versioning standard.

Goals

Predictable versioning is critical to enable multiple country implementations to share a common code base and derive shared value. This is a major goal of the 3.0 Re-Architecture. For example, Country A's implementation might fix a bug or add a new report, they would contribute that code to the open source project, and Country B could use it; and Country B could contribute something that Country A could use. For this to succeed, multiple countries using the OpenLMIS version 3 series must be upgrading to the latest Patch and Minor releases as they become available. Each country shares their bug fixes or new features with the open source community for inclusion in the next release.

Pre-Releases

Starting with version 3, OpenLMIS supports pre-releases following the Semantic Versioning standard.

Currently we suggest the use of **beta** releases. For example, 3.0 Beta is: 3.0.0-beta.

Note: the use of the hyphen consistent with Semantic Versioning. However a pre-release SHOULD NOT use multiple hyphens. See the note in **Modifiers** on why.

Modifiers

Starting with version 3, OpenLMIS utilizes build modifiers to distinguish releases from intermediate or latest builds. Currently supported:

Modifier: SNAPSHOT **Example**: 3.0.0-beta-SNAPSHOT **Use**: The SNAPSHOT modifier distinguishes this build as the latest/cutting edge available. It's intended to be used when the latest changes are being tested by the development team and should not be used in production environments.

Note: that there is a departure with Semantic Versioning in that the (+) signs are not used as a delimiter, rather a hyphen (-) is used. This is due to Docker Hub not supporting the use of plus signs in the tag name.

For discussion on this topic, see this thread. The 3.0.0 semantic versioning and schedule were also discussed at the Product Committee meeting on February 14, 2017.

We Prefer Coordination over Branching

Because each component is independently, semantically versioned, the developers working on that component need to coordinate so they are working towards the same version (their next release).

Each component's repository has a version file (gradle.properties or version.properties) that states which version is currently being developed. By default, we expect components will be working on the master branch towards a Patch release. The developers can coordinate any time they are ready to work on features (for a Minor release).

If developers propose to break with past API compatibility and make a Major release of the component, that should be discussed on the Dev Forum. They should be ready to articulate a clear need, to evaluate other options to avoid breaking backwards-compatibility, and to document a migration path for all existing users of the software. Even if the Dev Forum and component lead decide to release a Major version, we still require automated schema migrations (using Flyway) so existing users will have their data preserved when they upgrade.

Branching in git is discouraged. OpenLMIS does not use git-flow or a branching-based workflow. In our typical workflow, developers are all contributing on the master branch to the next release of their component. If developers need to work on more than one release at the same time, then they could use a branch. For example, if the component is working towards its next Patch, such as 1.0.1-SNAPSHOT, but a developer is ready to work on a big new feature for a future Minor release, that developer may choose to work on a branch. Overall, branching is possible, but we prefer to coordinate to work together towards the same version at the same time, and we don't have a branch-driven workflow as part of our collaboration or release process.

Code Reviews and Pull Requests

We expect all code committed to OpenLMIS receives either a review from a second person or goes through a pull request workflow on GitHub. Generally, the developers who are dedicated to working on OpenLMIS itself have commit access in GitHub. They coordinate in Slack, they plan work using JIRA tickets and sprints, and during their ticket workflow a code review is conducted. Code should include automated tests, and the ticket workflow also includes a human Quality Assurance (QA) step.

Any other developers are invited to contribute to OpenLMIS using Pull Requests in GitHub at any time. This includes developers who are implementing, extending and customizing OpenLMIS for different local needs.

For more about the coding standards and how to contribute, see **contributionGuide.md**.

Future Strategies

As the OpenLMIS version 3 installation base grows, we expect that additional strategies will be needed so that new functionality added to the platform will not be a risk or a barrier for existing users. Feature Toggles is one strategy the technical community is considering.

Rolling a Release

Below is the process used for creating and publishing a release of each component as well as the Reference Distribution (OpenLMIS 3.X.Y).

Goals

What's the purpose of publishing a release? It gives us a specific version of the software for the community to test drive and review. Beta releases will be deployed with demo data to the UAT site uat.openlmis.org. That will be a public, visible URL that will stay the same while stakeholders test drive it. It will also have demo data and will not be automatically wiped and updated each time a new Git commit is made.

Prerequisites

Before you release, make sure the following are in place:

- Demo data and seed data: make sure you have demo data that is sufficient to demonstrate the features of this release. Your demo data might be built into the repositories and used in the build process OR be prepared to run a one-time database load script/command.
- Features are completed for this release and are checked in.
- All automated tests pass.
- Documentation is ready. For components, this is the CHANGELOG.md file, and for the ref-distro this is a Release Notes page in the wiki.

Releasing a Component (or Updating the Version SNAPSHOT)

Each component is always working towards some future release, version X.Y.Z-SNAPSHOT. A component may change what version it is working towards, and when you update the serviceVersion of that component, the other items below need to change.

These steps apply when you change a component's serviceVersion (changing which -SNAPSHOT the codebase is working towards):

- Within the component, set the **serviceVersion** property in the **gradle.properties** file to the new -SNAPSHOT you've chosen.
  - See Step 3 below for details.
- Update **openlmis-ref-distro** to set **docker-compose.yml** to use the new -SNAPSHOT this component is working towards.
  - See Step 5 below for details.
  - Use a commit message that explains your change. EG, "Upgrade to 3.1.0-SNAPSHOT of openlmis-requisition component."
- Update **openlmis-deployment** to set each **docker-compose.yml** file in the deployment/ folder for the relevant environments, probably uat_env/, test_env/, but not demo_env/
  - See Step 7 below for details.
  - Similar to above, please include a helpful commit message. (You do not need to tag this repo because it is only used by Jenkins, not external users.)
- Update **openlmis-contract-tests** to set each **docker-compose...yml** file that includes your component to use the new -SNAPSHOT version.
  - Similar to the previous steps, see the lines under "services:" and change its version to the new snapshot.
  - You do not need to tag this repo. It will be used by Jenkins for subsequent contract test runs.
- (If your component, such as the openlmis-service-util library, publishes to Maven, then other steps will be needed here.)

Releasing the Reference Distribution (openlmis-ref-distro)

When you are ready to create and publish a release (Note that version modifiers should not be used in these steps - e.g. SNAPSHOT):

1. Select a tag name such as '3.0.0-beta' based on the numbering guidelines above.
2. The service utility library should be released prior to the Services. Publishing to the central repository may take some time, so publish at least a few hours before building and publishing the released Services:
   (a) Update the serviceVersion of GitHub's openlmis-service-util
   (b) Check Jenkins built it successfully
   (c) At Nexus Repository Manager, login and navigate to Staging Repositories. In the list scroll until you find **orgopenlmis-NNNN**. This is the staged release.
   (d) Close the repository, if this succeeds, release it. More information.
   (e) Wait 1-2 hours for the released artifact to be available on Maven Central. Search here to check: https://search.maven.org/
   (f) In each OpenLMIS Service's build.gradle, update the dependency version of the library to point to the released version of the library (e.g. drop 'SNAPSHOT')
3. In each service, set the **serviceVersion** property in the **gradle.properties** file to the version you've chosen. Push this to GitHub, then log on to GitHub and create a release tagged with the same tag. Note that GitHub release tags should start with the letter "v", so '3.0.0-beta' would be tagged 'v3.0.0-beta'. It's safest to choose a particular commit to use as the Target (instead of just using the master branch, default). Also, when you create the version in GitHub check the "This is a pre-release" checkbox if indeed that is true. Do this for each service/UI module in the project, including the API services and the AngularJS UI repo (note: in that repo, the file is called version.properties, not gradle.properties). DON'T update the Reference Distribution yet.
   (a) Do we need a release branch? No, we do not need a release branch, only a tag. If there are any later fixes we need to apply to the 3.0 Beta, we would issue a new beta release (eg, 3.0 Beta R1) to publish additional, specific fixes.
   (b) Do we need a code freeze? We do not need a "code freeze" process. We will add the tag in Git, and everyone can keep committing further work on master as usual. Updates to master will be automatically built and deployed at the Test site, but not the UAT site.
   (c) **Confirm** that your release tags appear in GitHub and in Docker Hub: First, look under the Releases tab of each repository, eg https://github.com/OpenLMIS/openlmis-requisition/releases. Next, look under Tags in each Docker Hub repository. eg https://hub.docker.com/r/openlmis/requisition/tags/ . You'll need to wait for the Jenkins jobs to complete and be successful so give this a few minutes. Note: After tagging each service, you may also want to change the serviceVersion again so that future commits are tagged on Docker Hub with a different tag. For example, after releasing '3.1.0' you may want to change the serviceVersion to '3.1.1-SNAPSHOT'. You need to coordinate with developers on your component to make sure everyone is working on 'master' branch towards that same next release. Finally, on Jenkins, identify which build was the one that built and published to Docker/Maven the release. Press the Keep the build forever button.
4. In **openlmis-config**, tag the most recent commit with the tag version (including the 'v').
5. Update **docker-compose.yml** in **openlmis-ref-distro** with the release chosen
   (a) For each of the services deployed as the new version on DockerHub, update the version in the **docker-compose.yml** file to the version you're releasing. See the lines under "services:" → serviceName → "image: openlmis/requisition-refui:3.0.0-beta-SNAPSHOT" and change that last part to the new version tag for each service.
   (b) Commit this change and tag the openlmis-ref-distro repo with the release being made. Note: There is consideration underway about using a git branch to coordinate the ref-distro release.

6. In order to publish the openlmis-ref-distro documentation to ReadTheDocs:
   (a) Edit **collect-docs.py** to change links to pull in specific version tags of README files. In that script, change a line like `urllib.urlretrieve("https://raw.githubusercontent.com/OpenLMIS/openlmis-referencedata/master/README.md", "developer-docs/referencedata.md")` to `urllib.urlretrieve("https://raw.githubusercontent.com/OpenLMIS/openlmis-referencedata/v3.0.0/README.md, "developer-docs/referencedata.md")`
   (b) To make your new version visible in the "version" dropdown on ReadTheDocs, it has to be set as "active" in the admin settings on readthedocs (admin -> versions -> choose active versions). Once set active the link is displayed on the documentation page (it is also possible to set default version).
7. Update **docker-compose.yml** in **openlmis-deployment** for the UAT deployment script with the release chosen which is at https://github.com/OpenLMIS/openlmis-deployment/blob/master/deployment/uat_env/docker-compose.yml
   (a) For each of the services deployed as a the new version on DockerHub, update the version in the **docker-compose.yml** file to the version you're releasing.
   (b) Commit this change. (You do not need to tag this repo because it is only used by Jenkins, not external users.)
8. Kick off each **-deploy-to-uat** job on Jenkins
   (a) **Wait** about 1 minute between starting each job
   (b) **Confirm** UAT has the deployed service. e.g. for the auth service: http://uat.openlmis.org/auth check that the **version** is the one chosen.
9. Navigate to uat.openlmis.org and ensure it works

Once all these steps are completed and verified, the release process is complete. At this point you can conduct communication tasks such as sharing the URL and Release Announcement to stakeholders. Congratulations!

Contributing

OpenLMIS is an open source community which appreciates the work of its contributors. Through contribution we're able to build a knowledgable community and make a wider impact than we would apart.

Contributing takes work so these guides aim to make that work clear and manageable:

Contributing to OpenLMIS

By contributing to OpenLMIS, you can help bring life-saving medicines to low- and middle-income countries. The OpenLMIS community welcomes open source contributions. Before you get started, take a moment to review this Contribution Guide, get to know the community and join in on the developer forum.

The sections below describe all kinds of contributions, from bug reports to contributing code and translations.

Reporting Bugs

The OpenLMIS community uses JIRA for tracking bugs. This system helps track current and historical bugs, what work has been done, and so on. Reporting a bug with this tool is the best way to get the bug fixed quickly and correctly.

Before you report a bug

- Search to see if the same bug or a similar one has already been reported. If one already exists, it saves you time in reporting it again and the community from investigating it twice. You can add comments or explain what you are experiencing or advocate for making this bug a high priority to fix quickly.
- If the bug exists but has been closed, check to see which version of OpenLMIS it was fixed on (the Fix Version in JIRA) and which version you are using. If it is fixed in a newer version, you may want to upgrade. If you cannot upgrade, you may need to ask on the technical forums.
- If the bug does not appear to be fixed, you can add a comment to ask to re-open the bug report or file a new one.

Reporting a new bug

Fixing bugs is a time-intensive process. To speed things along and assist in fixing the bug, it greatly helps to send in a complete and detailed bug report. These steps can help that along:

1. First, make sure you search for the bug! It takes a lot of work to report and investigate bug reports, so please do this first (as described in the section Before You Report a Bug above).
2. In the Description, write a clear and concise explanation of what you entered and what you saw, as well as what you thought you should see from OpenLMIS.
3. Include the detailed steps, such as the Steps in the example below, that someone unfamiliar with the bug can use to recreate it. Make sure this bug occurs more than once, perhaps on a different personal computer or web browsers.
4. The web browser (e.g. Firefox), version (e.g. v48), OpenLMIS version, as well as any custom modifications made.
5. Your priority in fixing this bug
6. If applicable, any error message text, stack trace, or logging output
7. If possible and relevant, a sample or view of the database - though don't post sensitive information in public

Example Bug Report

```
Requisition is not being saved
OpenLMIS v3.0, Postgres 9.4, Firefox v48, Windows 10

When attempting to save my in-progress Requisition for the Essential Medicines␣
↪program for the reporting period of Jan 2017,
I get an error at the bottom of the screen that says "Whoops something went wrong".

Steps:

1. log in

2. go to Requistions->Create/Authorize

3. Select My Facility (Facility F3020A - Steinbach Hospital)

4. Select Essential Medicines Program

5. Select Regular type
```

```
6. Click Create for the Jan 2017 period

7. Fill in some basic requested items, or not, it makes no difference in the error

8. Click the Save button in the bottom of the screen

9. See the error in red at the bottom. The error message is "Whoops something went␣
↪wrong".

I expected this to save my Requisition, regardless of completion, so that I may␣
↪resume it later.

Please see attached screenshots and database snapshot.
```

Contributing Code

The OpenLMIS community welcomes code contributions and we encourage you to fix a bug or implement a new feature.

Coordinating with the Global Community

In reviewing contributions, the community promotes features that meet the broad needs of many countries for inclusion in the global codebase. We want to ensure that changes to the shared, global code will not negatively impact existing users and existing implementations. We encourage country-specific customizations to be built using the extension mechanism. Extensions can be shared as open source projects so that other countries might adopt them.

To that end, when considering coding a new feature or modification, please:

1. Review your feature idea with the Product Committee. They may help inform you about how other country needs overlap or differ. They may also consider including a new feature in the global codebase using the New Feature Verification Process or reviewing the Global vs. Project-Specific Features wiki.
2. Before modifying or extending core functionality, email the developer forum or contact the Technical Committee. They can help share relevant resources or create any needed extension points (further details below).

Extensibility and Customization

A prime focus of version 3 is enabling extensions and customizations to happen without forking the codebase.

There are multiple ways OpenLMIS can be extended, and lots of documentation and starter code is available:

- The Reference UI supports extension by adding CSS, overriding HTML layouts, adding new screens, or replacing existing screens in the UI application. See the UI Extension Architecture and Guide.
- The Reference Distribution is a collection of collaborative **Services**, Services may be added in or swapped out to create custom distributions.
- The Services can be extended using **extension points** in the Java code. The core team is eager to add more extension points as they are requested by implementors. For documentation about this extension mechanism, see these 3 READMEs: openlmis-example-extensions README, openlmis-example-extension module README, and openlmis-example service README.
- Extra Data allows for clients to add additional data to RESTful resources so that the internal storage mechanism inside a Service doesn't need to be changed.

- Some features may require both API and UI extensions/customizations. The Technical Committee worked on a Requisition Splitting Extension Scenario that illustrates how multiple extension techniques can be used in parallel.

To learn more about the OpenLMIS extension architecture and use cases, see: https://openlmis.atlassian.net/wiki/x/IYAKAw.

Extension Points

To avoid forking the codebase, the OpenLMIS community is committed to providing **extension points** to enable anyone to customize and extend OpenLMIS. This allows different implementations to share a common global codebase, contribute bug fixes and improvements, and stay up-to-date with each new version as it becomes available.

Extension points are simply hooks in the code that enable some implementations to extend the system with different behavior while maintaining compatibility for others. The Dev Forum or Technical Committee group can help advise how best to do this. They can also serve as a forum to request an extension point.

Developing A New Service

OpenLMIS 3 uses a microservice architecture, so more significant enhancements to the system may be achieved by creating an additional service and adding it in to your OpenLMIS instance. See the Template Service for an example to get started.

What's not accepted

- Code that breaks the build or disables / removes needed tests to pass
- Code that doesn't pass our Quality Gate - see the Style Guide and Sonar.
- Code that belongs in an Extension or a New Service
- Code that might break existing implementations - the software can evolve and change, but the community needs to know about it first!

Git, Branching & Pull Requests

The OpenLMIS community employs several code-management techniques to help develop the software, enable contributions, discuss & review and pull the community together. The first is that OpenLMIS code is managed using Git and is always publicly hosted on GitHub. We encourage everyone working on the codebase to take advantage of GitHub's fork and pull-request model to track what's going on.

For more about version numbers and releasing, see **versioningReleasing.md**.

The general flow:

1. *Communicate* using JIRA, the wiki, or the developer forum!
2. *Fork* the relevant OpenLMIS project on GitHub
3. *Branch* from the `master` branch to do your work
4. *Commit* early and often to your branch
5. *Re-base* your branch *often* from OpenLMIS `master` branch - keep up to date!

6. Issue a *Pull Request* back to the `master` branch - explain what you did and keep it brief to speed review! Mention the JIRA ticket number (e.g., "OLIMS-34") in the commit and pull request messages to activate the JIRA/GitHub integration.

While developing your code, be sure you follow the Style Guide and keep your contribution specific to doing one thing.

Automated Testing

OpenLMIS 3 includes new patterns and tools for automated test coverage at all levels. Unit tests continue to be the foundation of our automated testing strategy, as they were in previous versions of OpenLMIS. Version 3 introduces a new focus on integration tests, component tests, and contract tests (using Cucumber). Test coverage for unit and integration tests is being tracked automatically using Sonar. Check the status of test coverage at: http://sonar.openlmis.org/. New code is expected to have test coverage at least as good as the existing code it is touching.

Continuous Integration, Continuous Deployment (CI/CD) and Demo Systems

Continuous Integration and Deployment are heavily used in OpenLMIS. Jenkins is used to automate builds and deployments trigged by code commits. The CI/CD process includes running automated tests, generating ERDs, publishing to Docker Hub, deploying to Test and UAT servers, and more. Furthermore, documentation of these build pipelines allows any OpenLMIS implementation to clone this configuration and employ CI/CD best practices for their own extensions or implementations of OpenLMIS.

See the status of all builds online: http://build.openlmis.org/

Learn more about OpenLMIS CI/CD on the wiki: CI/CD Documentation

Language Translations & Localized Implementations

OpenLMIS 3 has translation keys and strings built into each component, including the API services and UI components. The community is encouraging the contribution of translations using Transifex, a tool to manage the translation process. Because of the micro-service architecture, each component has its own translation file and its own Transifex project.

See the OpenLMIS Transifex projects and the Translations wiki to get started.

Licensing

OpenLMIS code is licensed under an open source license to enable everyone contributing to the codebase and the community to benefit collectively. As such all contributions have to be licensed using the OpenLMIS license to be accepted; no exceptions. Licensing code appropriately is simple:

Modifying existing code in a file

- Add your name or your organization's name to the license header. e.g. if it reads `copyright VillageReach`, update it to `copyright VillageReach, <insert name here>`
- Update the copyright year to a range. e.g. if it was 2016, update it to read 2016-2017

Adding new code in a new file

- Copy the license file header template, LICENSE-HEADER, to the top of the new file.
- Add the year and your name or your organization's name to the license header. e.g. if it reads `Copyright © <INSERT YEAR AND COPYRIGHT HOLDER HERE>`, update it to `Copyright © 2017 MyOrganization`

For complete licensing details be sure to reference the LICENSE file that comes with this project.

Feature Roadmap

The Living Roadmap can be found here: https://openlmis.atlassian.net/wiki/display/OP/Living+Product+Roadmap
The backlog can be found here: https://openlmis.atlassian.net/secure/RapidBoard.jspa?rapidView=46&view=planning.nodetail

Contributing Documentation

Writing documentation is just as helpful as writing code. See Contribute Documentation.

References

- Developer Documentation (ReadTheDocs) - http://docs.openlmis.org/
- Developer Guide (in the wiki) - https://openlmis.atlassian.net/wiki/display/OP/Developer+Guide
- Architecture Overview (v3) - https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=51019809
- API Docs - http://docs.openlmis.org/en/latest/api
- Database ERD Diagrams - http://docs.openlmis.org/en/latest/erd/
- GitHub - https://github.com/OpenLMIS/
- JIRA Issue & Bug Tracking - https://openlmis.atlassian.net/projects/OLMIS/issues
- Wiki - https://openlmis.atlassian.net/wiki/display/OP
- Developer Forum - https://groups.google.com/forum/#!forum/openlmis-dev
- Release Process (using Semantic Versioning) - https://openlmis.atlassian.net/wiki/display/OP/Releases
- OpenLMIS Website - https://openlmis.org

Contribute documentation

This document briefly explains the process of collecting, building and contributing the documentation to OpenLMIS v3.

Build process

The developer documentation for OpenLMISv3 is scattered across various repositories. Moreover, some of the artifacts are dynamically generated, based on the current codebase. All that documentation is collected by a single script. In order to collect a new document to be able to include it in the developer documentation, it must be placed in the *collect-docs.py* script. The documentation is built daily and is triggered by a Jenkins job. It then gets published via ReadTheDocs at http://docs.openlmis.org. The static documentation files and the build configuration is kept on the

openlmis-ref-distro repository, in the *docs* directory. It is also possible to rebuild and upload the documentation to Read the Docs manually, by running the *OpenLMIS-documentation* Jenkins job.

Contributing

Depending on the part of the documentation that you wish to contribute to, a specific document in one of the GitHub repositories must be edited. The list below explains where the particular pieces of the documentation are fetched from, in order to be able to locate and edit them.

Developer docs - Services: The documentation for each service is taken from the *README.md* file located on that repository.

Developer docs - Style guide: This is the code style guide, located in the openlmis-template-service in file *STYLE-GUIDE.md*.

Developer docs - Testing guide: This is the document that outlines the strategy and rules for test development. It is located in the openlmis-template-service in *TESTING.md* file.

Developer docs - Error Handling: This document outlines how errors should be managed in Services and how they should be reported through API responses.

ERD schema: The ERD schema for certain services is generated by Jenkins. The static file that links to the schema is located together with the documentation and the schemas itself are built and kept on Jenkins as build artifacts. The link always points to the ERD schema of the latest, successful build.

UI Styleguide: The configuration of the styleguide is located on the openlmis-requisition-refUI. The actual Styleguide is generated by the Jenkins job and uploaded to the gh-pages branch on the same repository.

API documentation: This contains the link to the Swagger documentation for the API endpoints. It is built by the Jenkins job and kept as a build artifact, based on the content of the RAML file. The link always points to the API documentation of the latest successful build.

CHAPTER 2

Links:

- **Project Management**
  - Issue Tracking & Project Management
  - Wiki
- **Communication**
  - Slack
  - Developer Forum
  - Product Comittee Forum
  - Governance Comittee Forum
- **Development**
  - GitHub
  - DockerHub (Published Docker Images)
  - OSS Sonatype (Maven Publishing)
  - Code Review
  - Code Quality Analysis (SonarQube)
  - CI Server (Jenkins)
  - CD Server
  - UAT Server